Visual Environment for Gravitational waves data Analysis

v0r95

D. Buskulic

LAPP / Université de Savoie

I Introduction

VEGA (Visual Environment for Gravitational waves data Analysis) is designed to be a framework in which one can analyze data coming from interferometric gravitational wave detectors. This data is supposed to be in the Frame format, which was agreed upon between the two gravitational wave detection experiments VIRGO and LIGO. This data format is now also used by other such experiments.

VEGA is based on the ROOT framework developed at CERN by R. Brun and F. Rademakers. There are numerous other contributors to this work and I apologize for not citing them all.

VEGA is an environment containing :

- The entire Framelib (http://wwwlapp.in2p3.fr/virgo/FrameL), which is a set of functions written in C that enable access and manipulation of frame data files
- A set of Manager classes that makes the heart of the framework, especially doing the interface with the Framelib and Framelib objects (frames, frame files, frame vectors).
- A set of frame data handling classes that allows the user an easy access to data contained in a big number of files.
- A set of algorithm classes that defines a standard form for algorithms, thus enabling easy manipulation and inclusion of these algorithms. They can be used in drawing, multiprocessing, etc...

So VEGA extends ROOT in the field of signal processing. We hope to provide in the near future as many tools as possible, for example a set of standard algorithms as well as an interface to Matlab.

Tutorials written by Andrew Haas (University of Washington), Mark Pollack (RHIC) and Fons Rademakers in part inspired this document. Thanks to them.

I.1 What is ROOT ?

ROOT is a system for large-scale data analysis and data mining. It is being developed for the analysis of Particle Physics data, but can equally well be used in other fields where large amounts of data need to be processed.

ROOT is not a computer program, but instead is a set of over 300 C++ classes and an interactive C++ interpreter. So one can use it as a framework, just using existing classes and libraries in his programs. Or, one can use the interactive part of the interpreter to execute tasks defined by the user on the command line or run macros, which are in fact made of C++ code. Once the user is satisfied with his macro, he can compile it (since it is nothing else than C++) with a little extra effort and run it much faster.

I.2 Objectives of this document

This User's Manual is written with the beginner or casual user in mind. Therefore, the first chapter is an "interactive" one in the sense that the best way to read it is in front of a computer with a VEGA session opened.

Next chapters are more for reference.

People that will begin to work with VEGA will usually have a FORTRAN or C background. I will suppose that the user is a little bit familiar with C and I will introduce the basic C++ notions when needed. This is not intended to be a C++ guide, it's too huge of a task, but ROOT is a good environment to learn C++ and it would be a pity not to use it that way! So I will try to be as simple as possible from the C++ point of view. Each example will be followed by the explanation of the underlying C++ concepts used, enough to understand the various commands. I will also introduce some ROOT related specific concepts after each example.

All suggestions are welcome and you can reach me: buskulic@lapp.in2p3.fr.

I.3 Installation of VEGA/ROOT

The installation and build of the VEGA software is described in "Installation of ROOT and VEGA" on the VEGA site

I.4 How to find information ?

The main source of information for ROOT is ROOT main site http://root/cern.ch. There is a lot of information online, and a User's Manual. In particular, one can find all the source code of ROOT organized in a very nice way, and a searchable archive of the mailing list devoted to ROOT, roottalk. If you have a question, there is a slight chance that someone asked it before you!
For VEGA, the information can be found at the VEGA site:
http://wwwlapp.in2p3.fr/virgo/vega/

II An interactive session

In this first chapter, we will concentrate on giving the reader a feeling of the possibilities of the framework. We will use standard ROOT as well as VEGA specific examples.

The user actions will be written in bold characters.

II.1 Basic intrinsic operations

First of all, one needs to know how to exit from a program! The C interpreter has some raw commands that begin with a dot. These commands are used to do intrinsic operations, not related to any C/C++ code. The ones that you should know for the beginning are:

.q	quits the interactive session
.x	loads a macro and executes it

We will see other raw commands, as we need them, especially commands to debug scripts (step, step over, set breakpoint, etc...).

II.2 The interpreter CINT, graphical interaction through a few examples

II.2.1 First very simple example

First, go to the $ROOTSYS/tutorials and start the ROOT/VEGA interactive session:

```
your_prompt$ vega
```

The prompt that you see is "vega[]". In fact, you launched the very small vega executable, which is in $VEGA/$UNAME and which is linked with the ROOT libraries.

This program gives access via a command-line prompt to all available ROOT/VEGA classes/globals/functions. By typing C/C++ statements at the prompt, you can create objects, call functions, execute scripts, etc. For example type:

```
vega[] 1+sqrt(9)
(double)4.000000000000e+00
```

```
vega[] for (int i = 0; i<5; i++) printf("Hello %d\n", i)
```

```
Hello 0
Hello 1
Hello 2
Hello 3
Hello 4
vega[] .q
```

As you can see, if you know C or C++, you can use ROOT. No new command line or scripting language to learn.

Some comments may be made at this point. First, there is no need to put a semicolon at the end of a line. Well, you can put one if you want. This is only true for statements written on the command line. One assumes that the carriage return is enough to express the end of the line for the interpreter. This is part of the interpreter's extensions to C++ that were made to allow an easy interaction, while not at the expense of breaking the compatibility with C/C++ language.

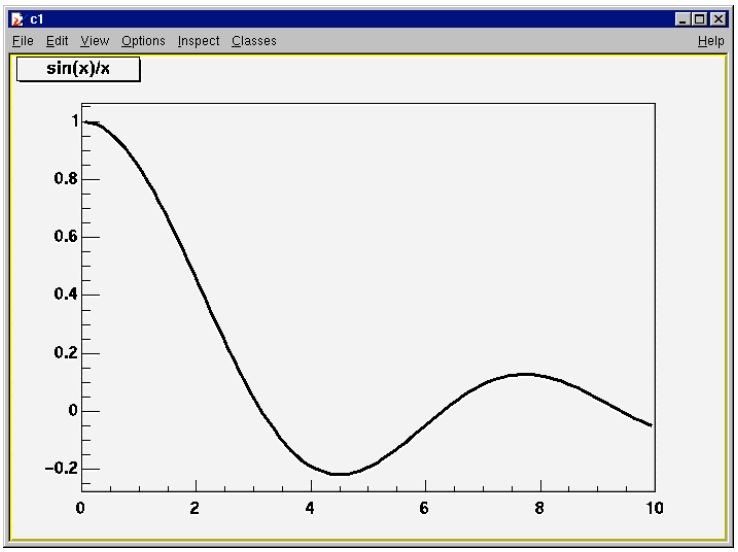
Some of the extensions don't work anymore inside a macro. For example, you have to put a semicolon at the end of each line in a macro.

II.2.2 Graphical output example

Let's try something more interesting. Again, start VEGA :

```
your_prompt$ vega


vega[] TF1 f1("func1", "sin(x)/x", 0, 10)
vega[] f1.Draw()
```

you should see something like this :



II.2.3 Classes, methods and constructors

We can stop for a second (well, a few seconds…) and explain what we have done. A more thorough introduction to C++ is given in “

More on C++
(for the interested reader) ” chapter.

In ROOT and C++, we introduce the notion of object, which is just a C structure with some internal functions (called “methods”) associated to it.

The line **`TF1 f1("func1", "sin(x)/x", 0, 10)`** created an object named **`f1`** of the type **`TF1`** which is a one-dimensional function. The type of an object is also called a class.

In fact, the line above builds an object by giving it a set of parameters. This is one example of a function associated to a class. It is a special one since it builds an object. It is called a constructor.

To interact with an object in ROOT, the usual syntax is :

object.action(parameters)

This is the usual way of doing in C++. The dot can be replaced by -> if object is a pointer. But since the interpreter always knows the type (class) of an object, if you put a dot instead, it will be automatically replaced when necessary.

So now, we understand the two lines that allowed us to draw our function.
`f1.Draw()` means “call the method Draw associated with the object f1 of class TF1”.
We will see the advantages of using objects and classes very soon.

One point : the ROOT framework was designed as an object oriented framework. This doesn’t mean that a user cannot call plain functions. For example all the FrameLib standard functions are available in the VEGA framework.

II.2.4 ROOT related specifics

We didn't yet explain everything about the constructor of f1 :

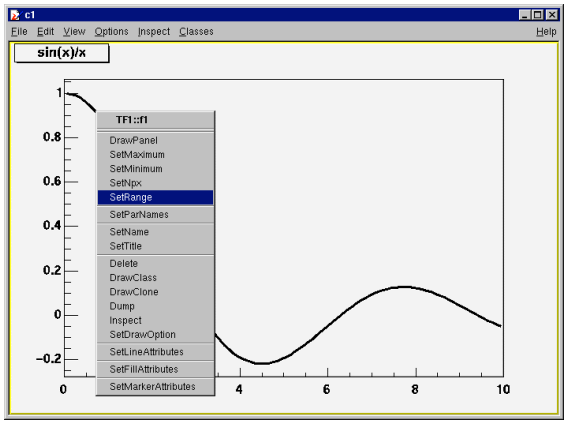
```
TF1 f1("func1", "sin(x)/x", 0, 10)
```

`"sin(x)/x"` is the function that we want to define, 0 and 10 the limits but what is `"func1"?` It's the name that we give to the object f1. Most objects in ROOT have a name. That way, there are lists of objects maintained by ROOT and it's easy to find an object by its name, especially if it is in a database. We will see why it is so when we see the notion of inheritance.

II.2.5 Graphical output example: User interaction

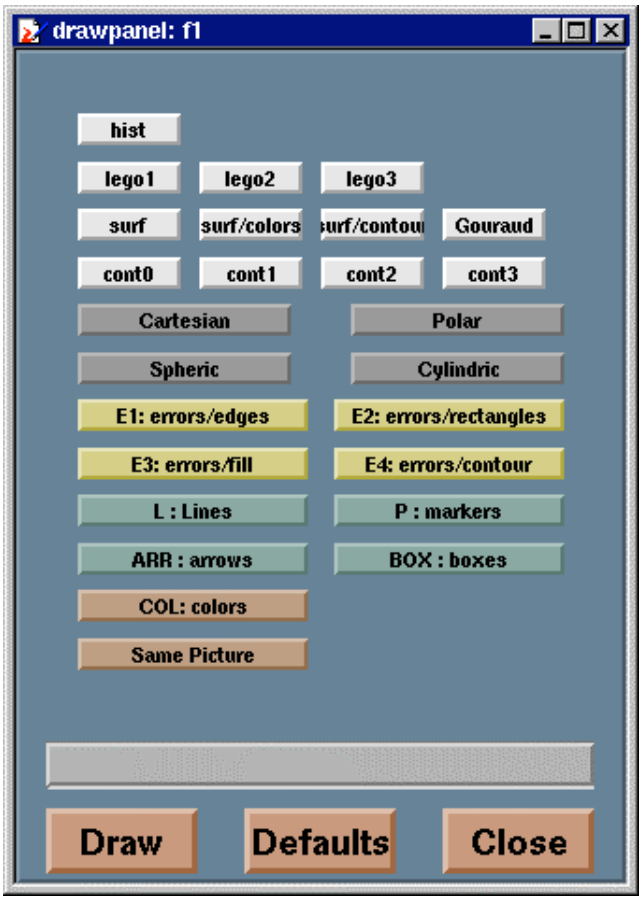
If you quit the framework, try to draw again the function `"sin(x)/x"`. Now, we can look at some interactive capabilities. Every graphics drawn in a window (which is called a Canvas) is in fact a graphical object in the sense that you can grab it, resize it, change some characteristics with a mouse click. And all the changes that you do are not done on a graphical copy of the object but on the object itself. For example try to click somewhere on the x-axis and drag along this axis. You have a very simple zoom.

When the cursor is on any object, you have access to selected methods by pressing the right mouse button and obtaining a context menu showing some available methods for this object. If you try this on the function itself, you obtain this kind of behavior:

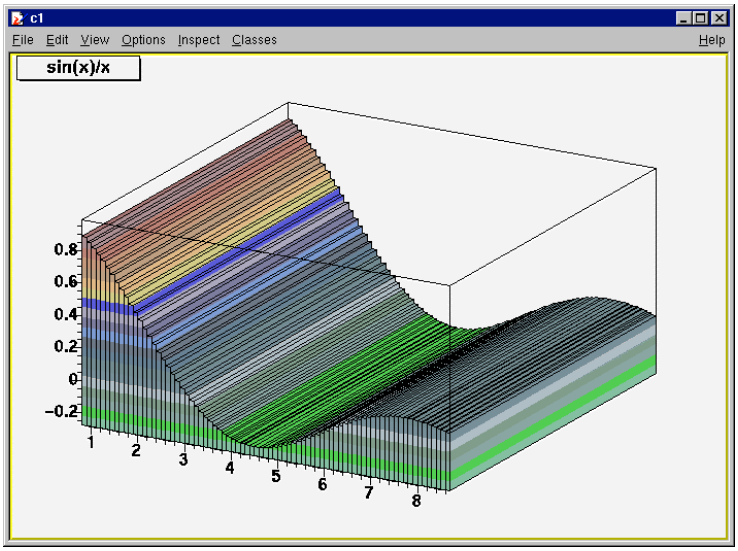


You can try for example to select the **`SetRange`** method and put -10, 10 in the dialog box fields. This is equivalent to executing the member function **`f1.SetRange(-10,10)`** from the command line prompt, followed by **`f1.Draw()`**.

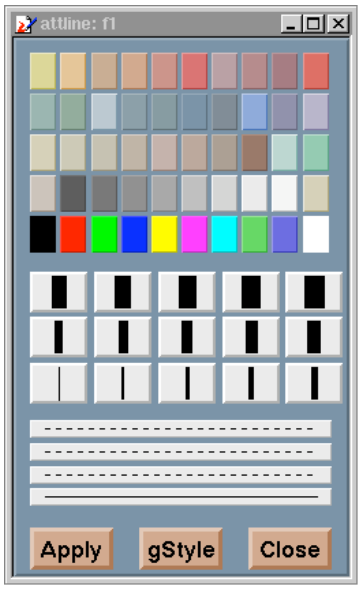
There are other things you may want to try. For example select the "DrawPanel" item of the popup menu. You will see a panel like this one :



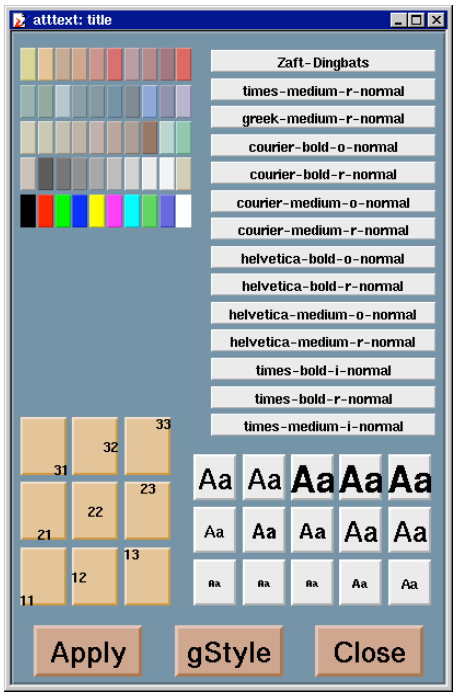
Try to resize the bottom slider and click Draw. You can zoom your graph. If you click on "lego2" and "Draw", you see a 2D representation of your graph :



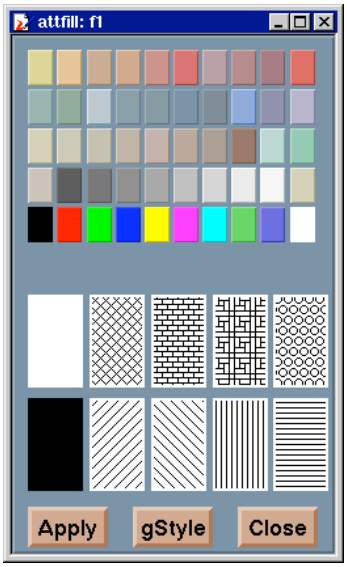
You can rotate interactively this 2D plot. Of course, it is possible to plot real 2D functions or graphs, not only a disguised 1D. There are numerous ways to change the graphical options/colors/fonts with the various methods available in the popup. Here are a few examples:



Line attributes



Text attributes



Fill attributes

Once the picture suits your wishes, you may want to see what are the lines of code you should put in a macro to obtain the same result. To do that, choose the "Save as canvas.C" option in the "Files" menu. This will generate a macro that you can look into to see how to set the various options. Notice that you can also save in postscript or gif formats the picture.

One other interesting possibility is to save in native root format your canvas. This will enable you to open it again and to change whatever you like, since all the objects associated to the canvas (histograms, graphs, etc...) are saved at the same time.

II.2.6 Second example : Building a multi-pad canvas

Let's now try to build a canvas (i.e. a window) with several Pads. The pads are sub-windows that can contain other Pads or graphical objects.

```
vega[] can = new TCanvas("can","Test canvas",1)
vega[] can.Divide(2,2)
```

Once again, we called the constructor of a class, this time the class TCanvas. The difference with the previous constructor call is that we want to build an object with a pointer to it. In C, we would have used malloc() or calloc(). Here, we use the operator **new**, which is the equivalent in C++. Instead of free() we will use **delete**. Usually, one has to declare a variable before using it. Since there cannot be any ambiguity in the line above, one doesn't need to declare a variable "can". In case such a statement in a macro is intended to be compiled, one would obviously have to declare the variable as a TCanvas* before using it.

Next, we call the method Divide() of the TCanvas class (that is TCanvcas::Divide()) which divides the canvas into 4 zones and sets up a Pad in each of them. Now, if we do

```
vega[] can.cd(1)
vega[] f1.Draw()
```

the function f1 will be drawn in the first Pad. All subsequent actions, draw an arrow for example, will be done on that Pad. To change the working Pad, there are three ways:

- Click on the middle button of the mouse on an object, for example a Pad. This sets this pad as the active one
- Use the method cd() of TCanvas (that is TCanvas::cd()) by telling the number of the pad, as was done in the example above:

```
vega[] can.cd(3)
```

Pads are numbered from left to right and from top to bottom.

- Each new pad created by the TCanvas::Divide method has a name, which is the name of the canvas followed by _1, _2, etc... for example to cd() to the third pad, you should do :

```
vega[] can_3.cd()
```

The third pad will set itself as the selected pad since you call the TPad::cd() method for the object can_3. This remark just to show the state of mind people have when they "think" in C++

II.2.7 Inheritance and data encapsulation

II.2.7.1 Inheritance

There is an obvious question that could be asked : what is the relation between a canvas and a pad ? Isn't it almost the same thing? Yes it is. In fact, a canvas **is a** pad that spans through an entire window. This is nothing else than the notion of inheritance. The TPad class is the parent of the TCanvas class.

So what? The most interesting thing is that you can reuse the code written in the parent class. For example, cd() is a method that TCanvas inherits from it's parent, TPad.

II.2.7.2 Method overriding

The method overriding is explained in chapter IV. Basically, when a class Child derives from another class Parent, it can reuse the methods defined in Parent, but if one method of Parent is not well adapted to Child, one can redefine (override) it. For example Draw() in some Child class may be different from Draw() in the Parent class. There is no ambiguity since you always specify the object : childobject.Draw() or parentobject.Draw().

II.2.8 ROOT related specifics

For what concerns inheritance, most objects in ROOT derive from a base class called TObject. This class contains everything that is shared by all the classes in ROOT. Especially, it handles

- the object I/O, so one can send any object to a file without writing the corresponding code,
- comparing two objects, enabling searches, sorts, etc…
- graphics hit detection
- notification between objects, so they can talk to each other…

to name the most important ones.

II.3 An example of a macro

Let's try now to see how we can write a simple macro. As we've said, this is just C/C++ code. Open the editor of your choice and look at the macro named vfill_histo.C. As a matter of convention, all macros are suffixed by .C. The source code has a .cxx or .cc suffix. vfill_histo.C is the following code :

```
{

// Create and fill a histogram with random #'s from a Gaussian distribution.
// See http://root.cern.ch/root/HowtoHistogram.html for a brief description
// of how to use histograms in ROOT.

     gROOT->Reset();

// TH1F is the equivalent of HBOOK1
// The fill data are 32 bit float points and weights are 64 bit fp.

     printf("Creating Histogram...\n");

     TH1F *h1 = new TH1F("h1","1-D Gaussians",100,-8,8);

     printf("done.\n");

//The ROOT/CINT extensions to C++. The declaration of h1 may be
//omitted when "new" is used.  So h1 = new TH1F(...) will correctly
//create an object of class "TH1F".

     gRandom->SetSeed();

// Raw C types are typedefed.
     Float_t xgauss;

     printf("Filling Histogram...\n");

     for (Int_t i=0; i < 10000; i++) {
          xgauss = gRandom->Gaus(3,0.5);
          h1->Fill(xgauss,0.2);
```

```
            h1->Fill(xgauss-6);
        }

        printf(" done.\n");

}
```

We can try to give detailed explanations of some of the lines above.

```
gROOT->Reset();
```

This statement resets the interpreter environment, it is almost always used at the beginning of a macro but not if you want to preserve some variables that a precedent macro defined. gROOT is a global variable. We will see some more details about globals in ROOT in a few moments. gROOT is an object of the class TROOT, which represents your session. It contains, among other things, lists of named objects created during the session, such as histograms. This enables access to those objects by their name.

```
TH1F *h1 = new TH1F("h1","1-D Gaussians",100,-8,8);
```

Here, we create a histogram by invoking one of its constructors. Walking through this statement, we have :

- TH1F: this is one of the histogram classes. All ROOT classes begin with a T (see Coding Conventions). As we will see, all VEGA classes begin with a V. H is for histogram, 1 for 1-D and F for using one float per channel.
- *h1 is the name of the pointer to this object
- new TH1F(...) invokes the constructor for this object with several parameters:
  - "h1" is the name of the histogram. In ROOT, as we have seen, most high level objects are named. Typically, one makes the name of the object the same as the pointer to which it refers.
  - "1-D Gaussians" is the title of the histogram. It will be displayed by default when the histogram is drawn.
  - 100, -8, 8 are the number of bins nbins, xlow and xup. For PAW users : you don't have to worry about the type of xlow and xup, integers will be converted to floats.

```
gRandom->SetSeed();
```

Another global. gRandom is the current object of the class TRandom of standard random generators. You probably already noticed that all globals begin with a "g"

followed by a capital letter. This is general and belongs to the coding conventions used in ROOT, which are followed by VEGA . The SetSeed() method of TRandom chooses a random seed, here the default one.

The class TRandom doesn't try to be overly ambitious. The important methods are :

- SetSeed(iseed) : sets the seed of the random number generator. If no number is given, 65536 is used.
- Gaus(mean, sigma) : return a random number distributed following a gaussian with mean and sigma.
- Rannor(a,b) : returns two random numbers following a gaussian with mean=0 and sigma=1.
- Rndm() : returns a uniformly distributed random number in the interval [0,1]

```
Float_t xgauss;
```

To avoid machine dependence of raw C types (double, float, and especially int, etc...), the basic types have been typedefed. So, instead of float, we use Float_t = 4 byte float. You can equally well use standard types (float, double) for your everyday work. In VEGA, we integrated the Framelib, which has its own machine-independent data types. To avoid problems, we decided to keep both. So you can use Float_t (ROOT type) or REAL_4 (Framelib type) indifferently. We know this can be misleading but one cannot avoid some historical background...

```
for (Int_t i=0; i < 10000; i++) {
    xgauss = gRandom->Gaus(3,0.5);
    h1->Fill(xgauss,0.2);
    h1->Fill(xgauss-6);
}
```

This seems to be a standard for loop in C in which you generate a random variable (gaussian) and you fill a histogram with this value.

For somebody not used to C++ something is strange : the Fill() method used to fill the histogram is used twice but not with the same number of arguments. This is perfectly common in C++, and it is called **method overloading**. We will see more about this in the next paragraph.

Now, to execute your macro type :

```
vega[] .x vfill_histo.C
Creating Histogram...
done
Filling Histogram...
```

```
done
```

Now you have in your memory (the computer one, not yours...) a histogram with which you can play. For example type

```
vega[] h1->Draw()
vega[] h1->Dump()
```

As will be seen in the chapter "Writing macros", one can write a named macro and give arguments to a macro. You also probably noticed that there is no includes in the macro. In fact, all the standard C includes are already done in the environment, as well as ROOT and VEGA specific ones.

II.3.1 C++ notions used: method overloading

We've just seen, in the example above, that a method can have different forms, with different arguments. If we look at an excerpt of the header file defining the class TH1F :

```
public:
     virtual void    Fill(Axis_t x);
     virtual void    Fill(Axis_t x, Stat_t w);
     virtual void    Fill(Axis_t x, Axis_t y, Stat_t w);
     virtual void    Fill(Axis_t x, Axis_t y, Axis_t z, Stat_t w);
```

You see that the Fill method is defined four times with different arguments. How does the compiler know what method to call ?

For the compiler to be able to make the difference, the arguments must be different enough. There is no ambiguity in the Fill methods above since the number of parameters is different for each method. If someone had made the following statement :

```
public:
     virtual void    Fill(Axis_t x);
     virtual void    Fill(Axis_t x, Stat_t w=1);
```

where in the second declaration, one of the arguments has a default value (this is permitted in C++), then the compiler doesn't know which method to call if someone uses

```
h1->Fill(23)
```

Should the compiler use the form `Fill(Axis_t x);` where x=23 or the form

`Fill(Axis_t x, Stat_t w=1);` where x=23 and one uses the default value w=1 ?

In that case, the compiler sends an error message. So one must be careful, when writing his methods not to come in such situations. Easier said than done.

II.4 How to compile a macro ?

Sometimes (in fact quite often) a script may become so big or be so processing-hungry that one would like to have it compiled and not interpreted. There is included into ROOT what is called a script compiler that allows compiling very easily the scripts you are going to produce.

It starts by calling the C++ compiler that was used to build ROOT and builds a shared library. This shared library is then loaded and the function defined in the script is executed.

II.4.1 Preparation of the script

So that seems nice, what is the deal ? It's only that you have to prepare a little bit your script. But it is quite simple. You have to

1) NOT use extensions to C/C++ provided by the CINT interpreter. That means for example that while in the interpreter you do not need to declare variables whose type is known from the context, this is mandatory if you want to be able to compile. Since the compiler is not supposed to know the context of the interpreter. An example. In the interpreter, you can use statements like :

```
c1 = new TCanvas("c1", "Test",1);
```

where you do not declare the variable c1, you should do instead :

```
TCanvas* c1 = new TCanvas("c1", "Test",1);
```

More generally, use very standard C/C++ and declare all your variables

2) You should add includes you use at the beginning of the file. Once again, this is for the real compiler that will do the work. Usually, the headers to be added are composed of the name of the classes used followed by the .h suffix. In case of included C code, the most commonly used are `"FrameL.h"` for all the Framelib calls, and the standard headers of the Virgo standard signal library.

3) In case you want to execute immediately (i.e. use the command ".x") the first function of the script you compile, you have to name the function in your script with the same name as the file it is contained in. For example the file my_test.C should begin with

```
my_test()
{
    your code….
}
```

The tutorial script called `scrollspectrum.C` has been prepared this way, check it.

II.4.2 Compiling the script

What remains to be done is to compile the script. If we want to execute the script called `scrollspectrum.C`, we just have to do (once in the `vegatutorial` directory) :

```
vega[] .x scrollspectrum.C++
```

notice the ++ at the end of the command. This tells the interpreter to compile `scrollspectrum.C`, build a shared library called `scrollspectrum.so`, load this shared lib and execute `scrollspectrum()`.

The resulting library scrollspectrum.so stays here and you can reuse it, that is reload it at a later time, with

```
vega[] .L scrollspectrum.so
```

furthermore, the scrollspectrum() function is available in the interpreter and may be used in some other script !

All this also works with loading the script after compiling :

```
vega[] .L scrollspectrum.C++
```

is a valid statement. This will build the library and load it as before. It will just not execute any function.

II.4.3 Compiling a more complex code

You will probably at one point or another try to compile a more complex code, consisting of a few files, or you will want to use the VEGA capabilities in your own code, for example use the display of vectors in a standalone executable. This is described in the chapter “How to compile your own code for VEGA or use the VEGA libraries in your code” at the end of the manual.

III Hands on VEGA

Before describing extensively the VEGA classes, methods and objects, it would be interesting to put directly hands on the environment and see some of it's capabilities from the point of view of gravitational waves data analysis. Concepts will be introduced as needed.

III.1 Preparation

So, let's imagine someone has a set of frame files that he/she wants to analyze. We will first build some tutorial files. Copy to your home directory the "vegatutorial" directory that is in the VEGA package. Then cd to this directory and launch a VEGA session. Then, in this session, execute the create_testfr.C script :

```
vega[] .x create_testfr.C
```

This will create files containing a few test frames, starting at time (GPS) 400000000.097, till the time 400001475.097. You will see some plots of the contents of these frames at the same time. Then, a file named SMSdemo.F containing frames with only slow monitoring test data will be produced.

III.2 Opening a Frame Channel

Now that we have a few frame files, we will need to access quickly a subset of them, if possible without bothering with frame boundaries, just giving the start time and length of the piece we want. In VEGA, the central notion to do such things is the "Frame Channel". Usually, the information about the data, called metadata (location, duration, triggers, etc...) is in one place and data itself is in another place. A frame channel will link the two, searching for information about data where it is and using it to access data. It will act as a data provider.

The metadata may take many forms. It may be a metadatabase, that will be described elsewhere, it may be a bookkeeping database, or it may be the frame files themselves that contain enough information to be used by a frame channel. We will start by using the simplest form, which is the direct use of the files.

Let's open such a frame channel with the demo frame files :

```
vega[] fc = new VFrameChannel("file:demo400000000.F demo400000045.F")
```

The operator new, described elsewhere, is creating an object of the class VFrameChannel, returning a pointer **fc**. The argument of the constructor (the function attached to the class that builds the objects of this class) are the names of

the frame files that will be used (in our case " demo400000000.F and demo400000045.F "). The characters "file:" are used to tell the frame channel which type of meta information it should use, in this case simple files.

NOTE : Since it's annoying to always type long names, one can use the tab completion feature of ROOT, using the <Tab> key. Here <Tab> represents the Tab key on the keyboard, NOT the letters "<.T.a.b.>.":

```
vega[] w = new VFrameC<Tab>
```

and the name is completed automatically. This is also true for names of files in the current directory. Even more, if you write :

```
vega[] w = new VFrameChannel(<tab>
```

You will have the list of variables used by this method, with the default parameters, and all the possible versions. This is VERY helpful.

One last word. Using a frame channel connected directly to files should be done only if frames are sequentially ordered in the files and not mixed inside several files. If the case is more complex, one should build a metadatabase and connect the frame channel to it. This is described later on.

III.3 Extracting a frame using a frame channel and plotting it

Now, as advertised, you can extract a frame from the set of files with :

```
vega[] fr = w->GetFrame(400000023)
```

To see the contents of the frame, one can do the standard Framelib call :

```
vega[] FrameDump(fr,stdout,2)
```

Which gives a summary of the contents of the frame. One can see that there is an ADC channel named MYGO_ADC1. But you can have the same kind of information from the framechannel itself by issuing

```
vega[] w->Print()
```

This will print all kinds of information about the framechannel and dump the contents of the first frame.

To draw the previously extracted frame in the current pad, just do :

```
vega[] gVM->Draw(fr,"adc.MYGO_ADC1")
```

Woaw ! what is this new beast, "gVM" ?

Well, if we wanted to be consistent with the "Object Oriented" way used in ROOT, we would have needed to use a fully C++ version of the Framelib. But this would have meant a complete change of the habits of many people. Since people are used to the C version, and since at the moment, the C++ version is not quite ready, we preferred to use a "manager class" that is taking the Framelib objects and building graphical objects with them. This class is called VManagerFrameL.

At initialization time, a global variable of type VManagerFrameL is build, called gVM

and it is available for all kinds of management tasks, taking care of drawn plots, freeing memory when necessary, taking care of a reference time, etc…

So, the last command we did created a plot (graphical representation, class VSPlot) of the frame `fr` content, more specifically of the adc channel named "MYGO_ADC1". The "adc" characters may be omitted if there is only one vector (adc proc or sim) with that name.

Even if it is supposed to be only a graphical representation of a vector, a VSPlot is much more than that. In fact, it contains almost all the information contained in a time series (FrVect for example). It can be the starting object for a truly "OO" way of doing time series.

For the moment, let's access the VSPlot object that was created :

```
vega[] vsp = gVM->GetLastPlot()
```

It was said that the VManager called gVM takes care of all the drawn plots. This instructs gVM to return us a pointer to the last drawn plot.

Now we can do things like

```
vega[] vsp->SetLineColor(4)
```

You can always use <Tab> to see the available methods. Furthermore, one can click on the drawn line or plot with the third mouse button, have a popup menu and select methods like we did for histograms. The interactive zoom is also available.

It is also possible to plot images coming from cameras that are included in a frame as FrVects in 2 dimensions. One uses the same syntax as for 1D, the manager gVM will take care of producing the right plot.

The tutorials contain a few examples of the use of gVM (see the macros scroll2.C and peak2.C for example).

III.4 Extracting a vector using a frame channel and plotting it

What if you don't want to take care of frame boundaries, just want to see a vector containing MYGO_ADC1 data, beginning at the time 400000023 and lasting 10 seconds ?

The frame channel can do that. Do :

```
vega[] ve = w->GetVect("MYGO_ADC1",400000023,10)
vega[] gVM->Draw(ve)
```

The first line extracts the vector, the second draws it. Notice that you didn't care about the frame boundaries, one of which is around 400000024.5, the other around 400000030.

Of course, one can extract a vector as big (or as small) as he wants, limited by the available machine memory of course.

One note : the user should take care of freeing the memory for all Framelib objects that are returned. Once a FrVect or a FrameH is extracted, it is the user's

responsibility to free memory in his loops with FrameFree(fr) and FrVectFree(ve). In a future version, we will try to do an automatic garbage collection, to free (!) the user from these tasks. For the moment, this is not so important since the amount of created vectors/frames is not big.

III.5 Extracting a 2D vector (image…) from the frame channel and plotting it

The vectors inside a frame may be two dimensional and may represent an image coming from a camera. 2D vectors are plotted the same way as 1D vectors. For example, assuming ve2 is a 2D vector (of type FrVect):

```
vega[] gVM->Draw(ve2)
```

will draw it.

Let's see a concrete example, using the files contained or generated in the vegatutorial directory. First, open a frame channel

```
vega[] fc = new VFrameChannel("file:demo400000000.F demo400000045.F")
```

second, extract the first frame using this frame channel

```
vega[] fr = fc->GetNextFrame()
```

third, since the name of the vector containing the image is "MYGO_IMAGE", plot it

```
vega[] gVM->Draw(fr,"MYGO_IMAGE","col")
```

That's it ! The option "col" used for this drawing produces a color plot. Other options are shown in the chapter "Representing gravitational waves data"

A few tutorials explain how to do it more generally: "moveimage.C", "tifrdisplay.C" for example.

III.6 Building a metadatabase for more complex tasks

In case one uses files containing frames that are not sequentially ordered or wants to use a selection or condition mechanism for the frames, the frame channel does not get enough information with a direct file access as described above. There is a necessity to build a metadatabase that will contain information about file location, triggers, quality words, etc… and that will be used by the frame channel as an index to access data. The details will be explained later but we can already build such a metadatabase for the tutorial files that we generated:

```
vega[] vd = new VFrameMetaDB("demoDB.root","CREATE","demo*")
```

The first argument of the constructor for a metadatabase is the name of the ROOT file that will contain metadata, in our case "demoDB.root", next the mode of opening ("READ" by default, but we used "CREATE" to build a new database). The third argument is the path to the files. As can be seen, one can use wildcards to specify

frame files names. But one could as well specify a directory name. The search is by default recursive, which means you can set the beginning of the search to the top of a directory tree containing directories, etc.…

Now, we can try to connect a frame channel to an existing database :

```
vega[] w = new VFrameChannel("demoDB.root")
```

The instructions to extract a frame or vector are the same as in the direct file access case. The only difference is that one is assured not to have any problems in case frames are mixed in various files or one wants to use a selected (triggered) set of frames.

III.7 Extracting a frame with a condition or selection

Frames are generally build with a certain set of conditions or selection parameters, coming from the online triggers or decided by the user. A user may want to process only a subset of the frames, corresponding to some conditions, say amplitude of the trigger “Trig1” higher than a certain value.

When building a metadatabase, the trigger information is read from the frame and saved. This allows for a frame channel, when using a metadatabase, to quickly browse it searching for frames that obey a given set of conditions.

Let’s illustrate the use of this feature with our tutorial test files. We will suppose that **w** is the frame channel opened in the previous paragraph. The frames contained in the tutorial files were generated with two trigger structures Trig1 and Trig2, which have various amplitudes for the various frames.

First, we may try to see the value of a given trigger amplitude by plotting it. The database object w has the capability to draw it:

```
vega[] w->DrawCondition("Trig1.amp","Trig1")
```

This will plot the amplitude of the trigger Trig1 provided it is defined. The second string "**Trig1**" is a selection expression, so one can plot only the amplitude of selected triggers if necessary.

The second step is to define the selection expression, with a set of conditions, which will govern the search for specific frames. This is done by defining a new object, called a condition set (class VConditionSet), that will contain the selection expression and serve as a pointer to a specified position in the metadatabase. Each type of metadatabase needs it's own type of condition sets. So we have to ask it, through the frame channel, to create a suitable condition set:

```
vega[] cs = w->CreateConditionSet("Trig1.amp>50 && Trig2>0")
```

The selection expression may use standard C operators and the main C math functions. "**sin(Trig1.amp)>0.23 || Trig2==0**" is a valid expression.

Now, one calls repeatedly the `GetNextFrame(cs)` method of `VFrDataBase` to get all the frames corresponding to the selection. To illustrate this, let’s load the next selected frame and see it’s beginning time :

```
vega[] fr = w->GetNextFrame(cs)
vega[] fr->GTimeS
(unsigned int)400000120
```

Do not forget that fr is a FrameH structure and as such, we can use its internal variables, such as GTimeS, the start time.

You see that the start time is obviously far from the start time of the first frame (400000001). All the frames in between were skipped since they did not obey the selection expression.

If you use repeatedly GetNextFrame, do not forget to delete the object fr after it's use (`with FrameFree(fr)`), otherwise your memory will get filled by all the undeleted frames that were passed to you.

III.8 How to deal with slow monitoring data ?

Slow monitoring data spans a lot of frames. Thus, it requires a special treatment. To illustrate this, let's build a database with frame files containing only SMS data. One such file was build by the create_testfr macro. Its name is SMSdemo.F. Check that you have such a file. Then, open a frame channel with for example:

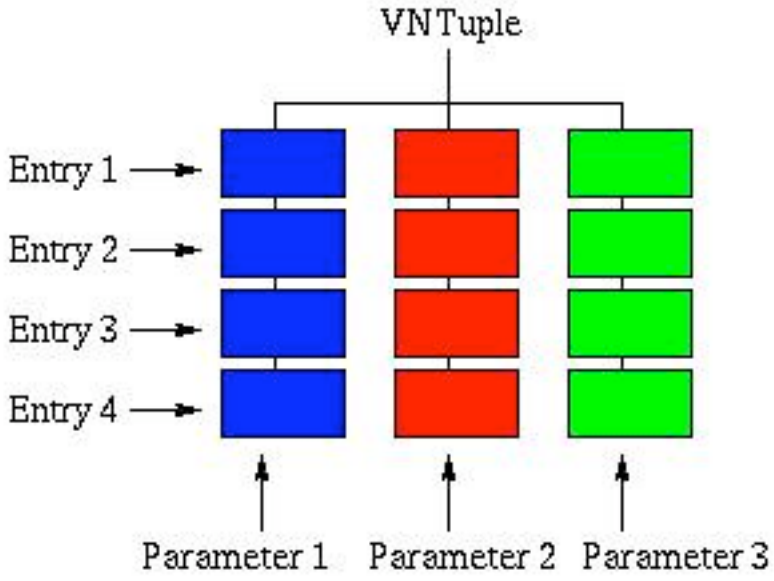
```
vega[] v = new VFrameChannel("file:SMSdemoDB.root")
```

If you want to see the contents of the frames, do as before :

```
vega[] v->Print()
```

You will see that there is a slow monitoring station called MYGOSMS that contains two variables VF1 and VI2.

The special treatment we have to apply in order to gain a simple yet efficient way of dealing with SMS data is to build a special object called an ntuple that will contain all this data. We developed a particular ntuple that differs from the standard ROOT one in that it is adapted to our needs. You can think of a ntuple as a list of all the sms data put in a tree-like structure :



The difference with a simple array is that each leaf of the tree can be any kind of object, even a tree itself. This leads to a hierarchy structure, like in a directory structure for a file system.

In fact, these more general ntuples are called trees and we will not yet work with them. In simple ntuples, each leaf is a single float parameter.

So to build a ntuple starting from the slow monitoring data, one can do :

```
vega[] nt = v->ExtractSMS("nt","MYGOSMS.VF1:MYGOSMS.VI2",
400000000,5000)
```

The first parameter **"nt"** is the name of the ntuple. Then, **"MYGOSMS.VF1:MYGOSMS.VF2"** is the list of SMS variables one wants to extract. Instead of telling every variable, you can specify only the name of a slow monitoring station, MYGOSMS in our case. The last two parameters are the start time from which we want the extraction and the length, expressed in seconds.

This ntuple will have three variables, the time being the first variable automatically inserted during extraction. If you want to see the contents of the ntuple, do :

```
vega[] nt->Print()
```

Now that we have a ntuple, one can easily plot all sort of things. For example, if you want to plot the variations of VF1 with time :

```
vega[] nt->DrawGraph("t:MYGOSMS.VF1","","at")
```

The second, empty parameter is a selection parameter. One can overlay a second plot :

```
vega[] nt->DrawGraph("t:MYGOSMS.VI2","","l")
```

One can also use some complicated expressions, for example

```
vega[] nt->DrawGraph("t:sqrt(MYGOSMS.VF1+MYGOSMS.VI2)","MYGOSMS.VF1 >
sqrt(MYGOSMS.VI2)+50","at")
```

Where you can see that the expressions can use the standard operators and simple

mathematical functions.

One can also plot a histogram of some columns of the ntuple :

```
vega[] nt->Draw ("MYGOSMS.VF1:MYGOSMS.VI2","","lego2")
```

See the result and try to rotate interactively the drawn histogram.

IV More on C++
(for the interested reader)

This paragraph is intended to introduce the reader to some useful insights into C++, to allow some use of the most advanced features found in ROOT. It is in no case a full course in C++.

IV.1 C++ notions used: classes, methods and constructors

You can try to use CINT to test the given examples. Before that, exit the previous ROOT session, go to the directory "$VEGA/vegatutorial", launch ROOT/VEGA and load into the environment the tutorial macro vtutorial.C :

```
vega[] .L vtutorial.C
```

The raw command .L loads a macro into the interpreter environment. You can then use all the functions/classes/objects defined inside it.

C++ extends C with the notion of class. If you're used to structures in C, a class is a struct, that is a group of variables that are related, which is extended with functions and routines specific to this structure (class). What is the interest? Consider a struct that is defined this way:

```
struct Line {
                              float x1;
                              float y1;
                              float x2;
                              float y2;
}
```

This structure is defined in vtutorial.C so you don't have to define it yourself. It represents a line to be drawn in a graphical window. (x1,y1) are the coordinates of the first point, (x2,y2) the coordinates of the second point.

In standard C, if you want to effectively draw such a line, you first have to define a structure and initialize the points (you can try this):

```
Line firstline;
```

```
firstline.x1 = 0.2;
firstline.y1 = 0.2;
firstline.x2 = 0.8;
firstline.y2 = 0.9;
```

This defines a line going from the point (0.2,0.2) to the point (0.8,0.9). To draw this line, you will have to write a function, say `LineDraw(Line l)` and call it with your object as argument :

```
LineDraw(firstline);
```

In C++, we would not do that. We would instead define a class like this:

```
class TLine {
                              int x1;
                              int y1;
                              int x2;
                              int y2;

                              TLine(int x1, int y1, int x2, int y2);
                              void Draw();
}
```

where we added two functions, that we will call methods or member functions, to the TLine class. The first method is used for initializing the line objects we would build. It is called a constructor.

The second one is the Draw method itself. So, to build and draw a line, we have to do:

```
TLine l(0.2,0.2,0.8,0.9);
l.Draw();
```

The first line builds the object ***l*** by calling its constructor. The second line calls the Draw() method of this object. You don't need to pass any parameters to this method since it applies to the object ***l*** and knows what are the coordinates of the line. These are internal variables x1, y1, x2, y2 that were initialized by the constructor.

IV.2 C++ notions used: inheritance and data encapsulation

IV.2.1 Inheritance

There is an obvious question that could be asked : what is the relation between a canvas and a pad ? Isn't it almost the same thing? Yes it is. In fact, a canvas **is a** pad that spans through an entire window. This is nothing else than the notion of

inheritance. The TPad class is the parent of the TCanvas class.

So what? The most interesting thing is that you can reuse the code written in the parent class. For example, cd() is a method that TCanvas inherits from it's parent, TPad. Let's try another example.

We've defined a TLine class that contains everything necessary to draw a line. If we want to draw an arrow, is it so different from drawing a line? We just have to draw a triangle at one end. It would be very inefficient to define the class TArrow from scratch. Fortunately, inheritance allows a class to be defined from an existing class. We would write something like :

```
class TArrow : public TLine {
     int ArrowHeadSize;

     void Draw();
     void SetArrowSize(int arrowsize);
}
```

The keyword "public" will be explained later. The class TArrow now contains everything that the class TLine does, and a couple of things more, the size of the arrowhead and a function that can change it. The Draw method of TArrow will draw the head and call the draw method of TLine. We just have to write the code for drawing the head!

IV.2.2 Method overriding

Giving the same name to a method (remember : method = member function of a class) in the child class (TArrow) as in the parent (TLine) doesn't give any problem. This is called **overriding** a method. Draw in TArrow overrides Draw in TLine. There is no possible ambiguity since, when one calls the Draw() method, this applies to an object which type is known. Suppose we have an object ***l*** of type TLine and an object ***a*** of type TArrow. When you want to draw the line, you do :

```
l.Draw()
```

so Draw() from TLine is called. If you do

```
a.Draw()
```

Draw() from TArrow is called and the arrow ***a*** is drawn.

IV.2.3 Public and private : data encapsulation

We have seen previously the keyword "public". This keyword means that every name declared public is seen by the outside world. This is opposed to "private" which means only the class where the name was declared private could see this name. For

example, suppose we declare in TArrow the variable `ArrowHeadSize` private as in :

```
private :
     int ArrowHeadSize;
```

. Then, only the methods (=member functions) of TArrow will be able to access this variable. Nobody else will see it. Even the classes that we could derive from TArrow will not see it. On the other hand, if we declare the method Draw() as public, everybody will be able to see it and use it. You see that the character public or private doesn't depend of the type of argument. It can be a data member, a member function, or even a class.

For example, in the case of TArrow, the base class TLine is declared as public :

```
class TArrow : public TLine {
```

This means that all methods of TArrow will be able to access all methods of TLine, but this will be also true for anybody in the outside world. Of course, this is true provided that TLine accepts the outside world to see its methods/data members. If something is declared private in TLine, nobody will see it, not even TArrow members, even if TLine is declared as a public base class.

What if TLine is declared "private" instead of "public"? Well, it will behave like any other name declared private in TArrow : only the data members and methods of TArrow will be able to access TLine, it's methods and data members, nobody else.

This may seem a little bit confusing and the reader should read a good C++ book if he/she wants more details. Especially since besides public and private, a member can be protected... But this is another story.

Usually, one puts private the methods that the class uses internally, like some utility classes, and that the programmer doesn't want to be seen in the outside world.

With "good" C++ practice (which we suppose is used in ROOT...), all data members of a class are also set private. This is called data encapsulation and is one of the strongest advantages of Object Oriented Programming (OOP). By doing that, nobody can see the data members of a class, except the class itself. So, from the outside world, if one wants to access those data members, one should use so called "getters" and "setters" methods, which are special methods used only to get or set the data members. The advantage is that if the programmer wants to modify the inner workings of his class, he can do so without changing what the user sees. The user doesn't even have to know that something has changed (for the better, hopefully).

An example. In our TArrow class, we would have set the data member `ArrowHeadSize` private. The setter method is `SetArrowSize()`, we don't need a getter method :

```
class TArrow : public TLine {
     private :
          int ArrowHeadSize;

     public :
          void Draw();
          void SetArrowSize(int arrowsize);
}
```

If one wants to define an arrow, he does :

```
TArrow* myarrow = new TArrow(1,5,89,124);
```

this will call the constructor of TArrow, but also the constructor of TLine, which is the parent class of TArrow, automatically. Then, we want to set the size of the head :

```
myarrow->SetArrowSize(10);
```

and draw it :

```
myarrow->Draw();
```

IV.3 C++ notions used: operators new and delete

Why is it so interesting to use new and delete instead of malloc and free ? First, the operator new knows the size of the object it has to initialize. You don't need to play with it. Second, and the most important, new calls the specified constructor of the class the object belongs to. No need to do special initialization, you have to think about it when you write your classes constructor.

But new also calls the constructor of all the objects that your class contains. For example, the class TPad contains a list of all objects in the pad. This list is itself an object of the class TList and there is a call to the constructor of this list in the constructor of TPad. We will see that new calls also automatically the constructor of the parent class (see the inheritance paragraph).

When you begin to use new, don't use malloc anymore. Better said, try to avoid mixing new and malloc. There is a risk to use free() instead of delete to release a memory that was allocated by new. This is guaranteed to crash!

V Accessing and managing Gravitational Waves data

V.1 Accessing frames through the Framelib

The Framelib is integrated into VEGA. This means that all the usual functions are accessible to the command line and in the macros. For example, you can try to run the macro vframel_ex.C. We let you compare it with the example FrameFull.c of the Framelib manual.

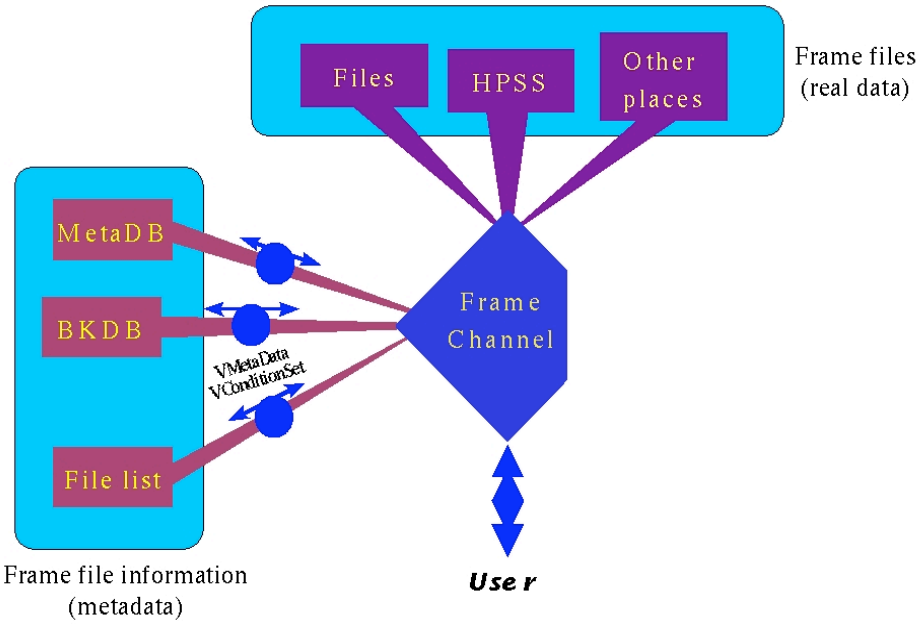
There is a way to draw the time series (vectors) extracted from frames, and this is described in the chapter "Representing gravitational wave data".

V.2 General scheme for data access

In a GW detection experiment, we will certainly end-up analyzing BIG amounts of data. Even if the format of these data is determined, the data itself will consist of many hundreds of files, each consisting of many megabytes. This is true even in local data that a user may want to analyze on his machine. Furthermore, when somebody uses some files locally, chances are that those files contain frames that are not contiguous, consisting of chunks of interesting data scattered through all the time the experiment was running. How then, access this data by time? Or by frame/run number?

The fundamental idea is to separate the files containing real data and the metadata, or data about data, containing information about frame file location, beginning and end times, conditions and triggers, etc… We will call the place where this metadata is stored an information database (may take various forms, as a metadatabase, a bookkeeping database, or the frame files themselves). A mechanism, called "frame channel" will act as a data provider, getting information about file location in the information database, extracting the data from the frame files and providing the result to the user.

This is shown in the following picture :



Except when building a metadatabase, the user interacts mainly with the frame channel to get the data in the form of frames or vectors.

The frame channel will interact with the information database through the exchange of metadata, while accessing directly the frame files. Clearly, this scheme is extendable to any kind of information or metadata and any type of frame location, even remote.

V.3 Different information databases for different uses

There are various types of information databases that are foreseen. Depending on the complexity of frame information, the user may want to choose one or the other.

V.3.1 Simple access: direct files

If one has a few files containing frames ordered sequentially, there is no need to build a database. One could think of accessing them with direct FrameLib calls. This is certainly doable. It may be convenient though to have exactly the same calling sequences than for more complex cases. One can think to build metadata on the fly with the information contained in the files themselves. So when building and opening a frame channel there is a direct files mode which is defined by adding the prefix "file:" to the name of the files one wants to open, as in :

```
vega[] fc = new VFrameChannel("file:demo400000000.F demo400000045.F")
```

The commands and functions to extract a frame or a vector are, from then on, the same as in the following cases.

V.3.2 More complex case: metadatabase

When the set of files becomes big, or the frames are scattered in non-sequential order in a set of files, or one wants to use conditions or triggers, it is necessary to build a metadatabase that will keep the information about frames location and characteristics. It will be used as an index to access the real frames. One builds such a metadatabase with, for example :

```
vega[] vd = new VFrameMetaDB("demoDB.root","CREATE","demo*")
```

The first argument of the constructor for a metadatabase is the name of the ROOT file that will contain metadata, in our case "demoDB.root", next the mode of opening ("READ" by default, but we used "CREATE" to build a new database). The third argument is the path to the files. As can be seen, one can use wildcards to specify frame files names. But one could as well specify a directory name. The search is by default recursive, which means you can set the beginning of the search to the top of a directory tree containing directories, etc.…

The metadatabase structure and methods are described in Appendix A.

When a metadatabase is build, one should connect a frame channel to it:

```
vega[] w = new VFrameChannel("demoDB.root")
```

The prefix "metadb" may be put in front of the metadatabase name, but this is the default, so it is not mandatory.

The instructions to extract a frame or vector are the same as in the direct file access case and will be explained in the following paragraphs. The only difference is that one is assured not to have any problems in case frames are mixed in various files or one wants to use a selected (triggered) set of frames.

V.3.3 Full case: bookkeeping database

In the case of VIRGO, a bookkeeping database is build continuously as frames come from the detector. A case is foreseen for connecting a frame channel to such a database. The prefix is "bkdb:". This will be described in future versions of VEGA.

V.4 Access to the data through a Frame Channel

V.4.1 Opening a frame channel and connecting to an information database

---

VFrameChannel(char* dbname)

To create a frame channel, we need to create an object of class VFrameChannel. At the same time, we can build it, looking for an information database. As, for example, in

```
vega[] fc = new VFrameChannel("demoDB.root")
```

we call the VFrameChannel constructor.

The parameter is the name of the info database. It may be preceded by the type of database :

- "file:" for direct file access. The frame files themselves will give the meta information.
- "metadb:" for a metadatabase, as described in the next paragraph
- "bkdb:" for an ORACLE bookkeeping database, feature not yet available, but in progress.

V.4.2 Accessing data through a Frame Channel

V.4.2.1 Extracting frames

Frames are extracted through the frame channel 'fc', opened as in the previous paragraph, with

```
FrameH* GetFrame(Double_t time)
FrameH* GetFrameR(Double_t relativetime)
```

The first method extracts a frame at an absolute time, the second at a time relative to the reference time (see chapter “Dealing with time”).

As in

```
vega[] FrameH* frame = fc->GetFrame(time)
```

where time is a double expressing a time that is contained in the frame. This method sends back a pointer to a well-known FrameH structure.

Once one has extracted one frame, it is possible to extract the next one or the preceding one in time :

```
FrameH* GetNextFrame()
```

For example :

```
vega[] frame = fc->GetNextFrame()
```

or

```
FrameH* GetPreviousFrame()
```

For example :

```
vega[] frame = fc->GetPreviousFrame()
```

Be careful, while GetFrame is checking to see if the requested time is really contained in one of the frames of the database, GetNextFrame and GetPreviousFrame do not. They will simply return you the frame that is the next (or previous) one in time in the database, even if it's years away.

V.4.2.2 Extracting vectors of any length

Sometimes, one would like to extract a vector which length is smaller than the one of a frame, or a vector that is spanning two or more frames. One can extract such a vector with :

```
FrVect* GetVect(Text_t* nameofvect, Double_t start, Double_t length,
        Option_t* opt="")
FrVect* GetVectR(Text_t* nameofvect, Double_t start, Double_t length,
        Option_t* opt="")
```

The first method extracts a frame at an absolute time, the second at a time relative to the reference time (see chapter “Dealing with time”).

As, for example :

```
vega[] FrVect* vect = fc->GetVect("adc.IFO_DMRO",start,length)
```

It returns a vector of type FrVect given its name and type. The vector starts at time "start" and has a length "length" (both are doubles). Vectors extracted through the frame channel will be concatenated as needed to obtain the desired vector. The string nameofvect indicates the type (adc, proc, sim) and the name of the series to be extracted. The convention for the format is "type.name".

For example "adc.IFO_DMRO" is a good format.

If only the name is given, the groups of series in the frame will be searched in the order adc, proc, sim for that name. If more than one identical name was defined in the frame, the first found series with that name will be used.

The extracted values of the vector are casted to double by default. This behaviour may be changed. If the option string `opt = "nocast"`, the extracted vector retains the original data type. “opt” is set by default to empty (no option).

V.4.2.3 Extracting n-tuples of SMS data

As was seen in the chapter “IV – Hands on VEGA”, in order to manipulate slow monitoring data, one has first to extract them in a ntuple. This is done with the VFrameChannel methods:

```
VNtuple* ExtractSMS(Text_t* nameofntuple, Text_t* varlist, Double_t
        start, Double_t length)
VNtuple* ExtractSMSR(Text_t* nameofntuple, Text_t* varlist, Double_t
        start, Double_t length)
```

The first method extracts an SMS at an absolute time, the second at a time relative to the reference time (see chapter “Dealing with time”).

- **nameofntuple** is the name of the new ntuple to be built.
- **varlist** is the list of variables to extract. Colons separate the variables, and each variable name is constructed with the name of the slow monitoring station, followed by the name of the variable, separated by a dot, like stationname.varname. So “TiServer.TA1:TiServer.T2:To9Tp.G41” is a valid variables list. If one wants to extract all the variables of a station, just give the name of the station, without any variable. “TiServer:To9Tp” will work (of course if these stations exist).
- **start** is the time from which the extraction starts in the database. It is an absolute GPS time for the **ExtractSMS** method and a time relative to the reference time for **ExtractSMSR**.
- **length** is the length of the extraction section, in seconds.

The variables and slow monitoring stations are checked for in the first frame accessible after the time **start**. If the variable names were given, the search will be made until the end of the searching section (till the time start+length is reached). But if all the variables of a station were asked for, and this station is not present, the ntuple will not be build.

V.4.3 Getting general information about the information database and it’s contents

V.4.3.1 Getting the start time of the information database

To get the start time of the first frame indexed in the information database (files, metadb, bookkeeping database) to which a frame channel is connected, one has to use the VFrameChannel method

```
double GetStart()
```

For example if fc is a valid frame channel,

```
vega[] fc->GetStart()
```

will output this start time while

```
vega[] st = fc->GetStart()
```

will put it in a double that may be reused.

V.4.3.2 Printing information about the frame channel

One can get some information about the frame channel and it's connected database by using the VFrameChannel method

---

```
void Print()
```

For example if fc is a valid frame channel,

```
vega[] fc->Print()
```

This will show a summary information about the information database, and will print the dump of the first frame at the end. This may be used to collect names of vectors, etc....

V.5 Dealing with selected or triggered data

V.5.1 Condition information in the information database

A frame channel is connected to an information database that may have various types. However, every such database is supposed to keep information about triggers, conditions and other such as quality words that allow a user to select a subset of all the frames. When the database is build by reading the frame files or getting information from them, the trigger information contained in each frame is gathered and arranged in such a way to allow easy extraction of frames or vectors satisfying some selection.

The triggers (structures of type FrTrigData in the FrameLib) are converted to more general objects called conditions. This will allow to use in the future as conditions other information such as slow monitoring data or quality information. The user may even add his own conditions, without copying the whole files just to add a FrTrigData structure.

The reader may find as an example the structure of the VEGA metadatabase in Appendix A

V.5.2 Condition Sets

Condition sets (class VConditionSet) are objects designed to be used as iterators on the conditions gathered in info databases, allowing a user to select a subset of all the conditions, through a selection formula, and to determine successively all the time intervals where this selection is valid. In other words, a condition set contains a selection expression on the conditions and is used as a pointer to the successive intervals of time satisfying this selection.

This will be clearer with an example. Suppose two triggers were defined in a set of

frames, say "Trig1" and "Trig2". These triggers have a given amplitude, which may vary, and we would like to select, for example, all time intervals for which "Trig1.amplitude>50 && Trig2.amplitude>1". This is our selection expression. A condition set will be build with this expression, as for example in:

```
vega[] cs = fc->CreateConditionSet("Trig1.amp>50 && Trig2.amp>1")
```

The object "fc" is a valid frame channel connected to an info database. Each information database has a particular type of condition set. So the frame channel will ask the info database to create a new condition set adapted to it's needs. All condition sets particular to a database derive from a common parent, called VConditionSet. This is the type of object returned by the CreateConditionSet() method.

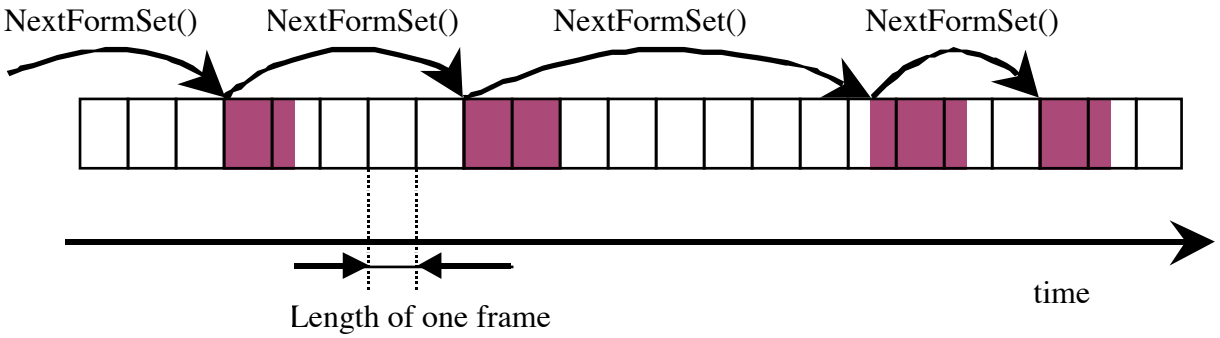
In other words, each database acts as a factory of condition sets that will be used only by itself and the frame channel.

The user may stop here since the extraction of all the successive frames or vectors satisfying the selection expression is described in the paragraphs below and do not need any other manipulation of the condition set. Nevertheless, for the interested reader, we can play with it, to understand the behavior.

When first build, a condition set points to the beginning of the info database. One can make it jump to the next set of valid conditions by using the method NextFormSet() :

```
vega[] cs->NextFormSet()
```

Using it repeatedly, the set will point to the successive selected time intervals, in dark in the following figure:



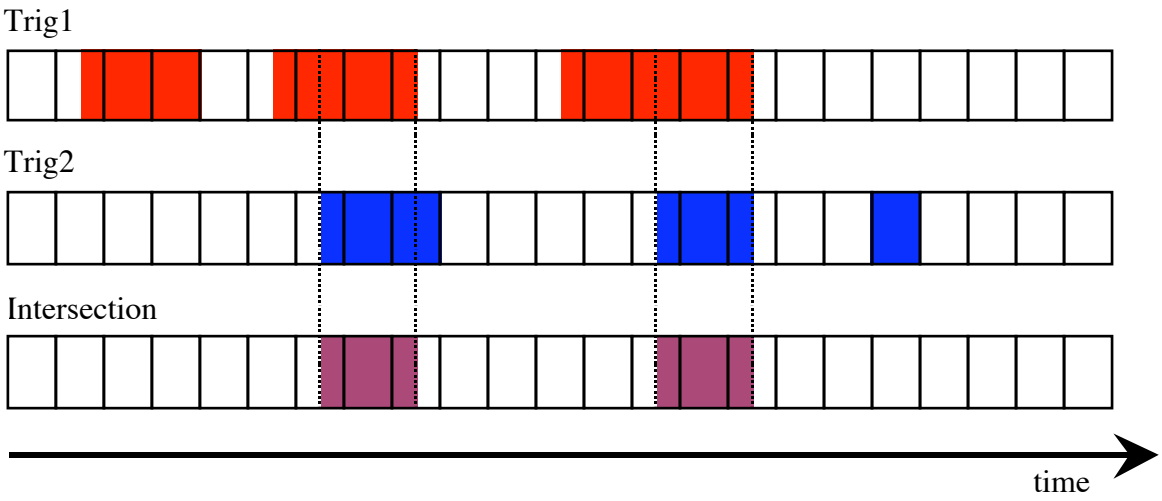
Actually, the selected intervals are not forced to have limits at the beginning or end of a frame. This depends on the start time and length of the triggers and is not dependent on the frame length.

Once the condition set has jumped to the next selected interval, one may retrieve its start and end time:

```
vega[] start = cs->GetIntersectionStart()
vega[] end = cs->GetIntersectionEnd()
```

Why is the word "Intersection" appearing here?

Each selection formula used in a condition set references one or more conditions (or triggers). For example, the formula above references "Trig1" and "Trig2". Each of these conditions has a start time and an end time. But in the case where there is more than one referenced condition, the start and end time of each of them may not coincide. So, when searching a match for the selection, the first thing done is to search for an intersection of all the referenced conditions:



In the example above, we have only used a sequential search, getting all the interesting intervals. It is also possible to jump directly to the interval that is following immediately a specified time :

```
vega[]cs->NearestGEFormSet(time)
```

where "time" is in GPS format (double, seconds.nanoseconds).

V.5.3 Extracting frames with a condition

There are two ways of extracting frames that satisfy a given condition: directly or sequentially. The direct method will be used if a particular frame, which approximate time is known, is to be extracted. The sequential method is used if one needs to process or view sequentially all the frames that correspond to a given selection expression.

V.5.3.1 Direct methods

The methods of VFrameChannel that allow a direct conditioned access to frames are:

`FrameH* GetNextFrame(Double_t time, char* selection)`

`FrameH* GetNextFrameR(Double_t relativetime, char* selection)`

The first method extracts a frame at an absolute time, the second at a time relative to the reference time (see chapter “Dealing with time”). "selection" is a selection expression referring to conditions existing in the database (hence the frames) such as "Trig1.amp>50 && Trig2.amp>2".

Example

```
vega[]frame = fc->GetNextFrame(time,"Trig1.amp>50&&Trig2.amp>2")
```

the search will begin from time "time". The frame returned will be the one containing the start time of the first interval satisfying the selection expression. This method sends back a pointer to a well-known FrameH structure.

V.5.3.2 Sequential methods

In order to access sequentially all the frames of interest, satisfying a selection expression, one has to have an object that will point to the intervals of interest. So came the idea of condition sets. Once a condition set has been defined, one can use it to extract frames that are recorded at the corresponding time. The methods of VFrameChannel to do so are:

```
FrameH* GetNextFrame(VConditionSet* condset)
FrameH* GetNextFrameR(VConditionSet* condset)
```

The first method extracts a frame at an absolute time, the second at a time relative to the reference time (see chapter “Dealing with time”). "condset" is a condition set defined as explained in the paragraph "Condition Sets"

Example

```
vega[]frame = fc->GetNextFrame(condset)
```

The search will be governed by the condition set "condset". The frame returned will be the one containing the start time of the next interval pointed to by "condset". These methods have to be called sequentially in a loop.

This method sends back a pointer to a well-known FrameH structure.

One warning: the memory for returned frames is allocated by the system, but it is the user's duty to free it after use. One has to do:

```
vega[]FrameFree(frame)
```

after finishing using the frame object.

V.5.4 Extracting vectors with a condition

Instead of extracting full frames, it is possible to extract only some vectors. The start time and length of the condition time interval will direct the start time and length of the vectors. The user does not (yet) control the length of the vector returned.

As for frames, the extraction may be direct or sequential.

V.5.4.1 Direct methods

The methods of VFrameChannel that allow a direct conditioned access to vectors are:

```
FrVect* GetNextVect(Text_t* nameofvect, Double_t gpstime, char*
        selection)
FrVect* GetNextVectR(Text_t* nameofvect, Double_t gpstime, char*
        selection)
```

They return a vector of type FrVect given its name and type. The vector spans a time interval for which the conditions expressed in the selection expression "selection" are valid. This time interval is the nearest next one to the time "gpstime".

Thus, "gpstime" is not the start time of the vector but the start time of the search.

The string nameofvect indicates the type (adc, proc, sim) and the name of the series to be extracted. The convention for the format is "type.name". For example "adc.IFO_DMRO" is a good format. If only the name is given, the groups of series in the frame will be searched in the order adc, proc, sim for that name. If more than one identical name was defined in the frame, the first found series with that name will be used.

The first method starts the search at an absolute time, the second at a time relative to the reference time (see chapter “Dealing with time”).

For example:

```
vega[]vect = fc->GetNextVect("adc.IFO_DMRO",gpstime,"Trig1>50")
```

Will return a vector of type FrVect,.

Frames where the condition expression "Trig1>50" will be first searched for. This will give a start time and length for the time interval of interest. Then, the vector of type adc and named "IFO_DMRO" will be assembled, whose start time and length will match those of the interval.

If only the name is given, the groups of series in the frame will be searched in the order adc, proc, sim for that name. If more than one identical name was defined in the frame, the first found series with that name will be used.

V.5.4.2 Sequential methods

In order to access sequentially all the vectors of interest, satisfying a selection expression, one has to have an object that will point to the time intervals of interest. So came the idea of condition sets. Once a condition set has been defined, one can use it to extract vectors that are recorded at the corresponding time. The method of VFrameChannel to do so is:

```
FrVect* GetNextVect(Text_t* nameofvect, VConditionSet* condset)
```

It returns a vector of type FrVect given its name and type. First, the condition set "condset" jumps to the next time interval satisfying its internal conditions. Then, the vector is extracted that spans this time interval.

The string nameofvect indicates the type (adc, proc, sim) and the name of the series to be extracted. The convention for the format is "type.name". For example "adc.IFO_DMRO" is a good format. If only the name is given, the groups of series in the frame will be searched in the order adc, proc, sim for that name. If more than one identical name was defined in the frame, the first found series with that name will be used.

For example:

```
vega[]vect = fc->GetNextVect("adc.IFO_DMRO",condset)
```

The search will be governed by the condition set "condset". The returned vector (of type FrVect) will be the next vector that satisfies the internal selection expression of the condition set "condset".

This method has been designed to be called sequentially in a loop.

One warning: the memory for returned vectors is allocated by the system, but it is the user's duty to free it after use. One has to do:

```
vega[]FrVectFree(vect)
```

after finishing using the vector object.

V.5.5 Getting information about condition sets

V.5.5.1 Printing the names of the conditions present in an information database

When printing information about a frame channel, the dump of the first frame is made. But this doesn't show all the conditions that exist in the connected database. To show the names of all the conditions gathered when building the info database, one has to use again the Print() method of VFrameChannel, but with an option "conditions". For example if fc is a valid frame channel:

```
vega[] fc->Print("conditions")
```

will do it.

V.5.5.2 Current status of a condition set

A condition set, as explained above, points to a particular section of data satisfying some criteria. It may be useful to get some information about the current status of a particular condition set. The following methods of VConditionSet are used to retrieve this information:

class VConditionSet

double GetIntersectionStart()

double GetIntersectionEnd()

Returns the start and end time of the current region of interest. This region is the intersection region of all the conditions that compose the condition set.

Example, provided cs is a valid condition set:

```
vega[] cs->GetIntersectionStart()
(double)661234567.1
vega[] cs->GetIntersectionEnd()
(double)661234569.3
```

class VConditionSet

double GetLastConditionResult()

Returns the result of the condition formula that is used to define the condition set applied to the current set of conditions. When searching for the next condition set, this result should be non zero.

class VConditionSet

double Eval(const char* formula)

Returns the result of the formula “formula” applied to the current set of conditions. This allows to see, in a set of conditions, the status of a particular condition. For example, if “fc” is a valid frame channel, suppose the specified condition set is :

```
vega[] cs = fc)>CreateconditionSet("Trig1.a>2 && Trig2.a<20")
```

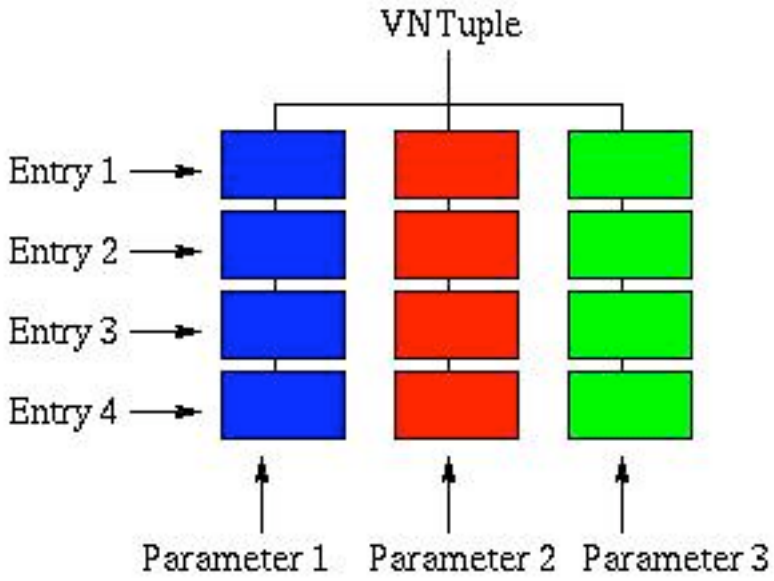
The fact that you jumped to the next valid condition set with a GetNextVect() or GetNextFrame() call (see VFrameChannel), ensures that the condition "**Trig1.a>2 && Trig2.a<20**" is satisfied. But you may wonder what is the real value of the amplitude of condition “Trig1”. This may be known with:

```
vega[] cs.Eval("Trig1.a")
(double) 11.876
```

V.6 N-tuples adapted for GW data analysis

Slow monitoring data spans a lot of frames. The special treatment we have to apply in order to gain a simple yet efficient way of dealing with SMS data is to build a special object called an ntuple that will contain all this data. We developed a particular ntuple (VNtuple) that differs from the standard ROOT one in that it is

adapted to our needs. You can think of a ntuple as a list of all the sms data put in a tree-like structure:



The difference with a simple array is that each leaf of the tree can be any kind of object, even a tree itself. This leads to a hierarchical structure, like in a directory structure for a file system.

In fact, these more general ntuples are called Trees. In simple ntuples, as VNtuples each leaf is a single float parameter.

VNtuples are derived from ROOT TNtuples and therefore, have access to all the methods available in TNtuples. We will focus on the ones that were added and on the ones most frequently used. For the others, the reader may refer to http://root.cern.ch/root/html/TNtuple.html.

V.6.1 Building VNtuple

The standard constructor is :

```
VNtuple(Text_t* name, Text_t* title, Text_t* listofvariables, Int_t
        bufsize=32000);
```

- **name** is the name of the new VNtuple.
- **title** is its title.
- **listofvariables** is the list of variables to be put in the ntuple. It is a char string made of names separated by colons. For example "t:x:y:z:var1". This list will tell what is the number of variables to be foreseen.
- **bufsize** is the buffer size used internally when writing to disk. The default value is sufficient for most uses.

Example :

```
vnt = new VNtuple("vnt","Example vntuple","t:TP1:TP2:PR1:PR2")
```

will build a new ntuple object with 5 variables. In a compiled program, vnt has to be declared as a VNtuple* before use. It will be automatically declared if one uses the interpreter.

V.6.2 Filling an N-tuple and getting data

The ExtractSMS method of VFrameChannel does the job for you in case you want to extract SMS data. It is nevertheless useful to know how to fill an ntuple and access the data that is inside it, in case you want to loop on this data.

```
void Fill(Float_t* x)
void Fill(Float_t x0, Float_t x1, Float_t x2=0, Float_t x3=0, Float_t
        x4=0, Float_t x5=0, Float_t x6=0, Float_t x7=0, Float_t x8=0,
        Float_t x9=0, Float_t x10=0, Float_t x11=0, Float_t x12=0,
        Float_t x13=0, Float_t x14=0)
```

- **x** is an array of floats.
- **x0 to x14** in case you have less than 15 variables and more than 1 (2 or more), you can call the second form of the Fill method.

Example :

```
vnt->Fill(1.43,4.23,b,c,d)
```

will fill the ntuple with the values specified.

One can fill an ntuple in a loop as much as he wants, especially if the ntuple is on disk. To make an ntuple on disk, simply open a ROOT file in “RECREATE” mode BEFORE building the ntuple. This one will automatically be attached to the file and it’s contents will be flushed to disk as soon as the buffers are full (every 32 kbytes by default).

```
void GetEntry(Int_t i)
Float_t* GetArgs()
```

Extracts the variables corresponding to a given entry in the ntuple.

- **i** is the index of the entry (in filling order).

GetEntry() fills an internal array with the contents of entry i and GetArgs() returns a pointer to an array of floats containing the values. There are two different methods because in the case of Trees, the objects to be returned may be very complex, and

one can try to get only a subset of these objects.

The **GetEntries()** method returns the number of entries in the ntuple.

A simple loop to process all the values of a ntuple would be:

```
while ( int i=0; i < vnt->GetEntries(); i++ ) {
   vnt->GetEntry(i);
   float* x = vnt->GetArgs();
   printf("nvar = %d, x0 = %fn",nvar,x[0]);
}
```

Of course, one can do any kind of treatment in the loop

If it's just a matter of printing the values, the Scan() method may be used. The previous example could be replaced by a simple **vnt->Scan()** call.

V.6.3 Drawing

The drawing methods are described in "Representing gravitational wave data".

VI Analysis of data coming from a frame channel: the selector

It is possible to analyze data coming from a frame channel by retrieving it sequentially or randomly. In this case, it is the user's responsibility to do the analysis loop. It may be interesting to automate the loop in order to let the user focus on the physics analysis, and in order to prepare for parallel processing.

This is done by using the frame channel `VFrameChannel::Process()` method which sends a restricted set of data from a frame channel to a processing script or class written by the user, called a "selector". Before we can process data, we need to make this selector.

Generation of a standard source skeleton, if "fc" is a valid frame channel, is made with the `VFrameChannel::MakeSelector` method :

```
vega[] fc->MakeSelector("selectorname");
```

This creates skeleton source code of a class named "selectorname" in two files `selectorname.h` and `selectorname.C`. The file `selectorname.C` is the only relevant for the user.

The generated class has a few initially empty methods that the user should fill :

- `void Begin(VFrameChannel *fc)` is called at the start of the processing. This is a good place to prepare the processing, such as booking the histograms.
- `Bool_t ProcessCut(FrameH* frame)` is called for each frame. If the result returned is "FALSE", the next steps of processing are ignored for this frame.
- `Bool_t ProcessFrame(FrameH* frame)` is called for each frame and does the real processing.
- `void Terminate()` is called at end of processing. One can display histograms here for example.

Once the file selectorname.C is customized, it becomes the argument of VFrameChannel::Process :

```
vega[] fc->Process("selectorname.C", 712345678.65,1300,"");
```

this will interpret the file selectorname.C and send it frames starting at GPS time 712345678.65, for a duration of 1300 s. The last argument is an optional condition on the data. One can also compile the script before processing :

```
vega[] fc->Process("selectorname.C++");
```

will compile the file selectorname.C, load it and process all the data available to the frame channel through it.

When appending a single "+", the file will be compiled and loaded, but subsequent compilations will only be done if the file has changed since the last compilation.

VII Representing Gravitational Waves data

VII.1 Series plots

Since the user may want to use the Framelib to access the frames, and since there are many structures defining time-series that can be used, there is a need for a class or set of classes defining a representation of a time-series or spectral series. There are two such classes, VSPlot for 1 dimensional plots and VSPlot2 for 2D.

The class VSPlot is the graphical representation of a series. For the time being, VSPlot derives from TH1D (histogram class) and adds some minor methods.

The drawing of VSPlot produces time series specific information such as the duration, mean value, etc... written in a statistical box.

As we will see, VSPlots are generally created and managed by graphical managers called VManagers. A VManager holds a potentially big number of plots and takes care of their deletion if necessary.

One can refer to examples in macros peak.C and peak2.C.

VII.2 Graphical managers of frames

VII.2.1 Definition of managers

The manager classes are the heart of the graphical part of the VEGA environment. They make the interface between the objects and libraries used by GW experiments, such as the framelib, frames and vectors, and the ROOT framework.

VManagerFrameL, is a class describing a manager that uses the Framelib for all its input/output of frames/vectors. We can have in the future a class named VManagerFramecpp that will use the frame class library Framecpp instead. It is not important to understand what VManagerFrameL exactly does, this will be explained in the next paragraph.

What do we want a general VManager to do ? Basically anything related to the interface between frames and VEGA. A first tentative list that will soon get longer may be:

- Drawing of vectors(1D and 2D)/series/images/
- Drawing of distributions of series/data
- Management of the VSPlots produced

- Etc…

VII.2.2 Building a manager : the global gVM

In VEGA, there is a global variable called gVM that is already declared as a VManagerFrameL and one can use it as he/she wants. gVM is the current VManager. If one declares another manager, gVM will point to it.

In practice, you can just forget about building a manager and use the standard gVM provided

If absolutely necessary, one starts a manager by building a new object of a VManagerXXX class. For example to start a manager using the FrameLib, one does :

```
vmgr = new VManagerFrameL();
```

VII.2.3 General use

Once building is done, which is by default already the case when you launch vega, one has to call methods from the declared gVM object. For example :

```
gVM->Draw(frame,"adc.IFO_DMRO");
```

will draw the data from the "IFO_DMRO" adc contained in the frame "frame" which is a FrameH structure loaded with standard Framelib calls. It will take care of creating a VSPlot or a VSPlot2 depending on the dimension of the vector.

We are going in the next paragraphs to concentrate on the manager that uses the Framelib. Almost the same routines will be available if another (C++ based) frame manager software is written and interfaced with VEGA.

VII.2.4 Drawing methods

VII.2.4.1 Drawing contents and distribution of frame vectors

The drawing methods of the VmanagerFrameL class are the following. They can be used with statements like

```
gVM->XXXX()
```

Where XXXX is the methods name:

```
void Draw(FrVect vect, Float_t offset, Float_t dur=1e20, Option_t
        *option="");
```

Will do a drawing of a vector (output on the screen is a VSPlot or VSPlot2) :

where

- **`vect`** is a frame vector of type FrVect that has been extracted from a frame.
- **`offset`** is the offset from which the drawing starts, with respect to the starting value of the vector (i.e. series).

- **dur** is the length of the drawn part of the vector (i.e. series).
- **option** is an option string that is passed to the plot that is drawn. This is equivalent to the TH1 class drawing options, except for one addition. The reader can go to the address http://root.cern.ch/ to see the list of options in the drawing of a TH1 (see Classes and Members Reference Guide). The added option is "sameti" option.

If **offset** and **dur** are omitted, the whole vector is drawn. It is also possible to specify only the drawing option.

A VSPlot is drawn in the current pad or canvas. The address of this plot is added to the list of plots managed by this manager. The name of the plot becomes the name of the FrVect.

Options :

- The ones of the TH1 class, for example "same" which superimposes a plot to a previous draw.
- "sameti" does a similar superposition but respects the time, i.e. will shift the plot so that it's time and the one of a previously drawn vector coincide.

Examples :

Suppose a time series vector **vect** contains data starting at t=0, and has a duration of 1 s, with 50000 samples.

gVM->Draw(vect) will draw the entire series.

gVM ->Draw(vect,0.3,0.2) will draw the series starting at t_0=0.3 s for a duration of 0.2 s, which represents 10000 samples.

gVM ->Draw(vect,"same") will draw the entire series on the same plot as a previous draw.

Drawing the distribution (histogram) of the data contained in an FrVect

---

```
void DrawHist(FrVect vect, Float_t offset=-1e20, Float_t dur=1e20,
              const Text_t *hwork="", Option_t *option="");
```

Makes a distribution of the values contained in a series. This method generates or uses a histogram of the class TH1F. If the histogram is generated, its limits are determined automatically to be +/- 5% of the max and min value in the series and this histogram has 100 bins.

The variables are :

- **offset** is the offset from which the histogramming starts, with respect to the starting value of the series.
- **dur** is the length of the histogramed part of the series.

- **hwork** is, optionally, the name of an existing histogram, for example "histo1". This allows using any limits for the histogram, provided the user defines one before calling DrawHist. The contents of the histogram are erased before filling by the new values, except if a "+" precedes the name, as in "+histo1". In this case, the data is appended to the histo. If the name hwork is given but there is no already existing histogram of that name, a new one is created with standard characteristics.
- **option** is an option string that is passed to the drawn histogram of type TH1F. This is equivalent to the TH1 class drawing options. The reader can go to the address http://root.cern.ch/ to see the list of options in the drawing of a TH1 (see Classes and Members Reference Guide).

Examples :

Let's assume that a time series **vect** contains data starting at t=0, and has a duration of 1 s, with 50000 samples. The min value is -123, the max value is 212.

```
gVM->DrawHist(vect,0.2,0.4)
```

will draw the histogram of the values in vect beginning at t_0=0.2 s for a duration of 0.4 s. The generated histogram has limits [-130,223].

```
h1 = new TH1F("h1","test histo",100,-25,25);
```

```
gVM->DrawHist(vect,0.2,0.4,"h1");
```

will generate the histogram **h1**, with limits [-25,25]. Calling the DrawHist method will fill this histogram but only with the values in its range. So one can concentrate on the distribution of values around 0.

```
gVM->DrawHist(vect,0.2,0.4,"+h1");
```

This will do the same thing as before but without erasing the values of the histogram before filling. It will just add those to the existing ones. That way, one can span more than one series for one given histogram.

VII.2.4.2 Drawing selected contents or distribution of a frame

One can draw a selected adc or processed or simulated data without having to extract it first from a frame.

Simple drawing of a vector (1D or 2D) contained in a frame :

```
void Draw(FrameH frame, Text_t* typeAndName, Float_t offset, Float_t
        dur=1e20, Option_t *option="");
```

where

- **frame** is a frame of type FrameH that has been read from a file.
- **typeAndName** is a pointer to an array of char that describes the type (adc, proc or sim) and name of the frame element to be extracted and drawn. It has the

following structure : "type.name". For example "adc.IFO_DMRO" is a good format. It will look for the adc data of the adc named "IFO_DMRO". It is possible to give only the name, if there is no ambiguity. In that case, adc data will be searched first for that name, then proc and then sim data. If more than one identical name was used, the first found series will be extracted.

- **offset** is the offset from which the drawing starts, with respect to the starting value of the vector (i.e. series).
- **dur** is the length of the drawn part of the vector (i.e. series).
- **option** is an option string that is passed to the plot that is drawn. This is equivalent to the TH1 class drawing options. The reader can go to the address http://root.cern.ch/ to see the list of options in the drawing of a TH1 (see Classes and Members Reference Guide).

Options :

- The ones of the TH1 class, for example "same" which superimposes a plot to a previous draw.
- "sameti" does a similar superposition but respects the time, i.e. will shift the plot so that it's time and the one of a previously drawn vector coincide.

If **offset** and **dur** are omitted, the whole vector is drawn. It is also possible to specify only the drawing option.

A VSPlot is drawn in the current pad or canvas. The address of this plot is added to the list of plots managed by this manager. The name of the plot becomes the name of the FrXXXData extracted (XXX = Adc, Proc or Sim).

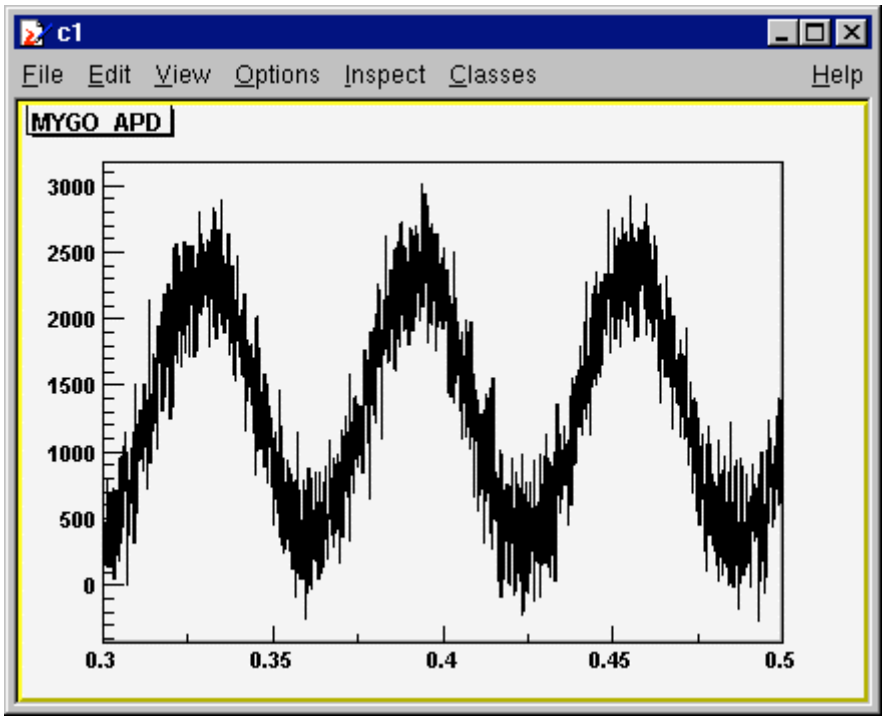
Examples :

Suppose a FrAdcData vector **"MYGO_ADC"** contains data starting at t=0, and has a duration of 1 s, with 50000 samples in a frame called "**frame**".

gVM->Draw(frame,"adc.MYGO_ADC") will draw the entire series.

gVM ->Draw(frame,"MYGO_ADC",0.3,0.2) will draw the series starting at t_0=0.3 s for a duration of 0.2 s, which represents 10000 samples. Since no type (adc proc or sim) was given, it will search for the first data occurring with this name in the frame.

Example of such a plot :



gVM ->Draw(frame,"MYGO_ADC","same") will draw the entire series on the same plot as a previous draw.

Drawing the distribution (histogram) of the data of a vector contained in a frame

---

```
void DrawHist(FrameH frame, Text_t* typeAndName, Float_t offset=-
        1e20, Float_t dur=1e20, const Text_t *hwork="", Option_t
        *option="");
```

Makes a distribution of the values contained in a vector of type FrXXXData (XXX = Adc, Proc or Sim) contained in the frame "frame" and of type and name **typeAndName**. This method generates or uses a histogram of the class TH1F. If the histogram is generated, its limits are determined automatically to be +/- 5% of the max and min value in the series and this histogram has 100 bins.

The variables are :

- **frame** is a frame of type FrameH that has been read from a file.
- **typeAndName** is a pointer to an array of char that describes the type (adc, proc or sim) and name of the frame element to be extracted and drawn. It has the following structure : "type.name". For example "adc.IFO_DMRO" is a good format. It will look for the adc data of the adc named "IFO_DMRO". It is possible to give only the name, if there is no ambiguity. In that case, adc data will be searched first for that name, then proc and then sim data. If more than one identical name was used, the first found series will be extracted.
- **offset** is the offset from which the histogramming starts, with respect to the

starting value of the series.

- **dur** is the length of the histogramed part of the series.
- **hwork** is, optionally, the name of an existing histogram, for example "histo1". This allows using any limits for the histogram, provided the user defines one before calling DrawHist. The contents of the histogram are erased before filling by the new values, except if a "+" precedes the name, as in "+histo1". In this case, the data is appended to the histo. If the name hwork is given but there is no already existing histogram of that name, a new one is created with standard characteristics.
- **option** is an option string that is passed to the drawn histogram of type TH1F. For the time being, this is equivalent to the TH1 class drawing options. The reader can go to the address http://root.cern.ch/ to see the list of options in the drawing of a TH1 (see Classes and Members Reference Guide).

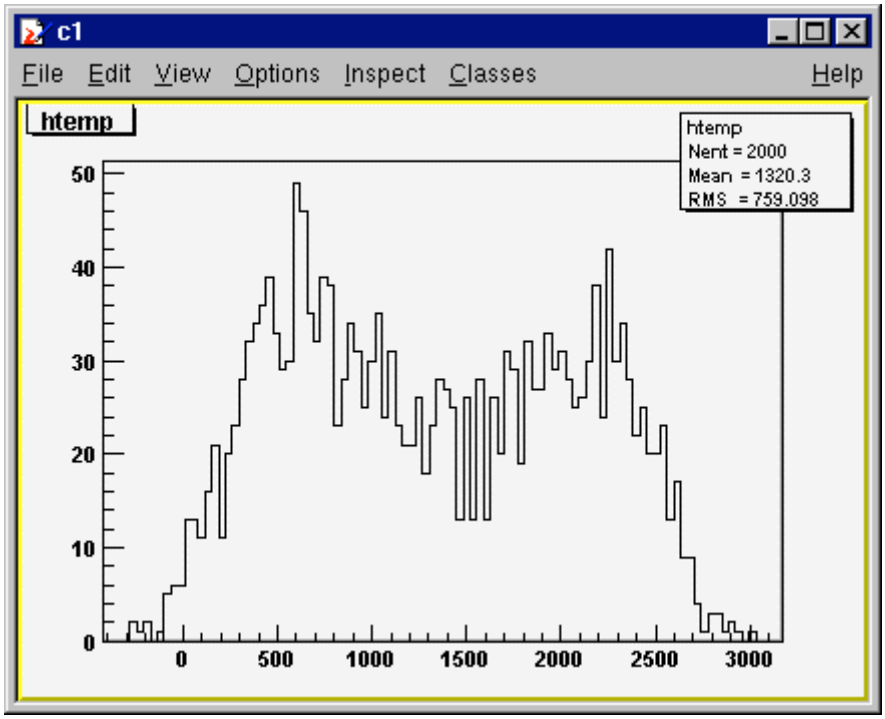
Examples :

Let's assume that a FrAdcData vector contains data starting at t=0, and has a duration of 1 s, with 50000 samples. The min value is -123, the max value is 212 in a frame called "**frame**".

```
gVM->DrawHist(frame,"adc.MYGO_ADC",0.2,0.4)
```

will draw the histogram of the values in the adc **"MYGO_ADC"** beginning at t_0=0.2 s for a duration of 0.4 s. The generated histogram has limits [-130,223].

Example of such a plot :




```
h1 = new TH1F("h1","test histo",100,-25,25);
```

```
gVM->DrawHist(frame,"MYGO_ADC",0.2,0.4,"h1");
```

will generate the histogram **h1**, with limits [-25,25]. Calling the DrawHist method will fill this histogram but only with the values in its range. So one can concentrate on the distribution of values around 0. Since no type (adc proc or sim) was given, it will search for the first data occurring with this name in the frame.

```
gVM->DrawHist(frame,"MYGO_ADC",0.2,0.4,"+h1");
```

This will do the same thing as before but without erasing the values of the histogram before filling. It will just add those to the existing ones. That way, one can span more than one series for one given histogram.

VII.2.4.3 Drawing 2D vectors (images or time-frequency plots)

Frame vectors may contain bi-dimensional information such as images coming from cameras or time-frequency plots. There is no special method for drawing these 2D plots. One uses gVM->Draw() methods as for 1D and these methods take care of producing the right plot, 1D or 2D depending on the dimension of the data.

In the case of 2D, the plot produced is of type VSPlot2. The manager handles a list of these plots, as well as a list of 1D plots.

Example :

In the files produced by the create_testfr.C script in the vegatutorial directory, some frames contain images. They are generated as a sum of two dimensional functions and are there only for demonstration purposes. Once in vegatutorial, the following lines should show an image:

first, open a database

```
vega[] vd = new VFrDataBase("demoDB.root")
```

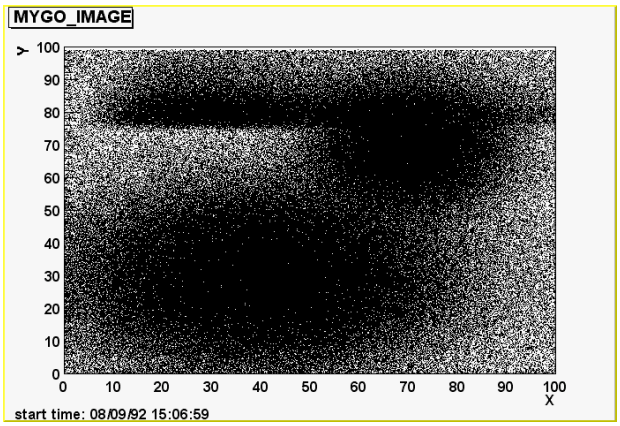
second, extract the first frame of this database

```
vega[] fr = vd->GetNextFrame()
```

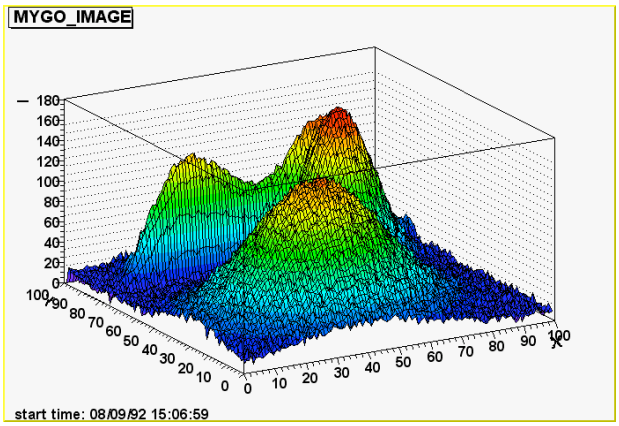
third, since the name of the vector containing the image is "MYGO_IMAGE", plot it

```
vega[] gVM->Draw(fr,"MYGO_IMAGE","col")
```

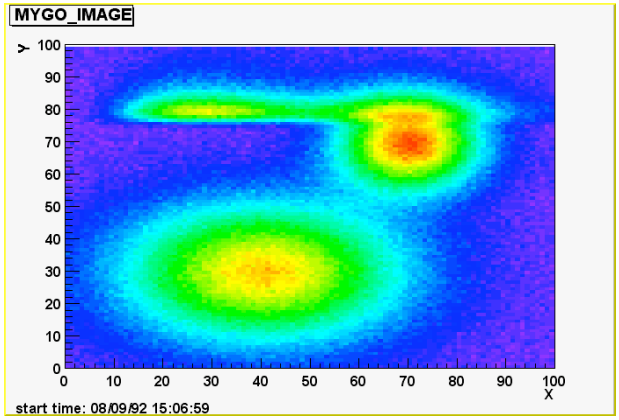
That's it ! The option "col" used for this drawing produces the third plot showed hereafter, other options are shown.



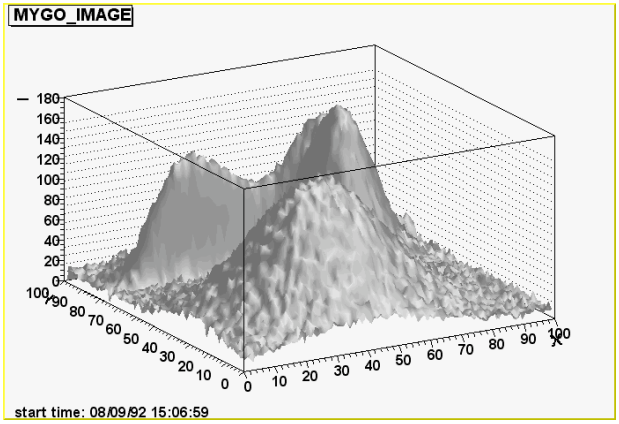
No option (scatter plot)



Option "surf1"



Option "col"



Option "surf4"

The options available are the ones of the ROOT TH2 class and are summarized below:

"SAME"	Superimpose on previous picture in the same pad
"CYL"	Use Cylindrical coordinates
"POL"	Use Polar coordinates
"SPH"	Use Spherical coordinates
"PSR"	Use PseudoRapidity/Phi coordinates
"LEGO"	Draw a lego plot with hidden line removal
"LEGO1"	Draw a lego plot with hidden surface removal
"LEGO2"	Draw a lego plot using colors to show the cell contents
"SURF"	Draw a surface plot with hidden line removal
"SURF1"	Draw a surface plot with hidden surface removal
"SURF2"	Draw a surface plot using colors to show the cell contents
"SURF3"	same as SURF with in addition a contour view drawn on the top
"SURF4"	Draw a surface using Gouraud shading
"CONT"	Draw a contour plot (same as CONT0)
"CONT0"	Draw a contour plot using surface colors to distinguish contours
"CONT1"	Draw a contour plot using line styles to distinguish contours
"CONT2"	Draw a contour plot using the same line style for all contours
"CONT3"	Draw a contour plot using fill area colors
"CONT4"	Draw a contour plot using surface colors (SURF option at theta = 0)
"LIST"	Generate a list of TGraph objects for each contour
"ARR"	Arrow mode. Shows gradient between adjacent cells
"BOX"	A box is drawn for each cell with surface proportional to contents
"COL"	A box is drawn for each cell, with a color scale varying with contents
"BB"	With LEGO or SURFACE, suppress the Back-Box
"FB"	With LEGO or SURFACE, suppress the Front-Box
"AH"	Draw plot, but not the axis labels and tick marks (for LEGO and SURF options)
"Z"	Draw a color scale on the right of the plot (for COL and CONT options only)
"SCAT"	Draw a scatter-plot (default)

Since an image is attached to a given frame and does not span two or more of them, one cannot, for the moment, extract a 2D vector spanning two or more frames. One could indeed think of it for time-frequency vectors but it is not yet done.

VII.2.5 Change of the plots attributes

Once drawn, it is possible to change the plots attributes like line color or fill style directly using gVM. The following methods are available :

GetXaxis(const char* name="")

GetYaxis(const char* name="")

GetZaxis(const char* name="")

Return a pointer to the specified axis object, allowing to manipulate it.

SetBarOffset(Float_t baroff = 0.5, const char* name="")

SetBarWidth(Float_t barwidth = 0.5, const char* name="")

SetFillColor(Color_t color = 1, const char* name="")

SetFillStyle(Style_t styl = 0, const char* name="")

SetLineColor(Color_t color = 1, const char* name="")

SetLineStyle(Style_t styl = 0, const char* name="")

SetLineWidth(Width_t width = 1, const char* name="")

SetLabelColor(Color_t color = 1, Option_t* axis = "X", const char* name="")

SetLabelFont(Style_t font = 62, Option_t* axis = "X", const char* name="")

SetLabelOffset(Float_t offset = 0.005, Option_t* axis = "X", const char* name="")

SetLabelSize(Float_t size = 0.04, Option_t* axis = "X", const char* name="")

SetMarkerColor(Color_t tcolor = 1, const char* name="")

SetMarkerSize(Size_t msize = 1, const char* name="")

SetMarkerStyle(Style_t mstyle = 1, const char* name="")

SetNdivisions(Int_t n = 510, Option_t* axis = "X", const char* name="")

SetTickLength(Float_t length = 0.03, Option_t* axis = "X", const char* name="")

The arguments are self explanatory, except the name. This is the name of the plot on which the change of attribute should be done. If name="", the change is applied to the last plot drawn by gVM.

Examples:

```
gVM->SetMarkerSize(1.2)
```

will change the marker size of the last plot drawn

```
gVM->SetLineColor(3,"MYGO_ADC1")
```

will change the line color of the plot named "MYGO_ADC1".

VII.2.6 Management of the plots produced

The manager decides at drawing time what is the dimension of the plot produced depending on the dimension of the data in the frame vector. If the frame vector is 1 dimensional, it produces a VSPlot, and if it is 2D, it produces a VSPlot2. The manager then handles two lists of produced plots, one for 1D and one for 2D.

One can get a pointer to the last created and plotted VSPlot:

```
VSPlot* GetLastPlot()
```

As in :

```
vega[] VSPlot* vs1 = gVM->GetLastPlot()
```

This could be useful if one wants to modify this VSPlot, for example to change some attributes. An example is given in the macro "peak2.C".

One can also get a pointer to the last created and plotted VSPlot2 :

```
VSPlot* GetLastPlot2()
```

As in :

```
vega[] VSPlot2* vs1 = gVM->GetLastPlot2()
```

This could be useful if one wants to modify this VSPlot2, for example to change some attributes.

VII.3 Representing slow monitoring data

There are three main powerful drawing methods for slow monitoring data, i.e. VNtuples. They all enable putting selections, i.e. drawing data that pass given cuts.

The first method draws VSPlots, which are standard representations of time series. This method is mainly used to treat slow monitoring data as simple time series. We encourage you to use this first method since it gives powerful manipulation (signal analysis) capabilities.

```
void DrawSeries(Text_t* varlist, Test_t* cuts, Text_t* option, Int_t
        nentries=1000000000, Int_t firstentry=0)
```

The parameters are almost identical to the ones of the second method, and are described below. The only difference is in the "option" parameter.

- **option** is an option for drawing:

"SAME"	Superimpose on previous picture in the same pad
"CYL"	Use Cylindrical coordinates
"POL"	Use Polar coordinates
"SPH"	Use Spherical coordinates
"PSR"	Use PseudoRapidity/Phi coordinates
"LEGO"	Draw a lego plot with hidden line removal
"LEGO1"	Draw a lego plot with hidden surface removal
"LEGO2"	Draw a lego plot using colors to show the cell contents
"SURF"	Draw a surface plot with hidden line removal
"SURF1"	Draw a surface plot with hidden surface removal
"SURF2"	Draw a surface plot using colors to show the cell contents
"SURF3"	same as SURF with in addition a contour view drawn on the top
"SURF4"	Draw a surface using Gouraud shading
"AH"	Draw plot, but not the axis labels and tick marks
"B"	Bar chart option
"C"	Draw a smooth Curve through the plot points
"E"	Draw error bars
"E0"	Draw error bars including bins with o contents
"E1"	Draw error bars with perpendicular lines at the edges
"E2"	Draw error bars with rectangles
"E3"	Draw a fill area through the end points of the vertical error bars
"E4"	Draw a smoothed filled area through the end points of the error bars
"L"	Draw a line through the plot points
"P"	Draw current marker at each point
"*H"	Draw plot with a * at each point

The second method draws graphs (sets of points defined by two coordinates). It will be used in the case of data sets irregularly spaced in time.

```
void DrawGraph(Text_t* varlist, Test_t* cuts, Text_t* option, Int_t
        nentries=1000000000, Int_t firstentry=0)
```

- **varlist** is an expression of the general form e1:e2 where e1,etc is a formula referencing a combination of the ntuple columns
  Example:
  varexp = x:y simplest case: draw a plot of points (x,y), x and y representing columns named x and y
  = t:sqrt(x) : draw plot of sqrt(x) vs t
  = log(t):x*y/z
  Note that the variables e1 or e2 may contain a selection.
  example, if e1= x*(y<0), the value plotted will be x if y<0 and will be 0 otherwise.
- **cuts** is a selection expression. Only ntuple entries passing this selection will enter into the graph. For example, it could be something like "t>10 && sin(TP1/6.28) <0". One can use standard C operators (+ - * / && || !...) and usual mathematical functions, including transcendental ones.
- **option** is an option for drawing. The available options are :

option = 'A'	Axis are drawn around the graph
option = 'T'	Time is used for axis labels (not a standard graph feature, added by VEGA)
option = 'P'	The current marker is drawn at each point
option = 'L'	A simple polyline between all the points is drawn
option = 'F'	A fill area is drawn ('CF' draw a smooth fill area)
option = 'C'	A smooth Curve is drawn
option = '*'	A Star is plotted at each point
option = 'B'	A Bar chart is drawn at each point

- Options should be combined to obtain the desired effect. Typically, the first graph is drawn with option 'AP', to draw the axis, and subsequent plots that the user wants to overlay are plotted with the 'P' option alone.
- **nentries** is the maximum number of entries to be processed in the ntuple
- **firstentry** is the first entry to be processed

The second method draws histograms. This is mainly used for statistical studies.

```
void Draw(Text_t* varlist, Test_t* cuts, Text_t* option, Int_t
        nentries=1000000000, Int_t firstentry=0)
```

Unlike in DrawGraph, which needs at least two variables to plot a graph, one can plot a 1D, 2D or 3D histogram. The variables have the same meaning as in DrawGraph() but the varlist can contain the name of an existing histo and options are not the same :

- **varlist** is an expression of the general form e1:e2 where e1,etc is a formula referencing a combination of the ntuple columns
  Example:
  varexp = x:y simplest case: draw a plot of points (x,y), x and y representing columns named x and y
  = t:sqrt(x) : draw plot of sqrt(x) vs t
  = log(t):x*y/z
  Note that the variables e1 or e2 may contain a selection.
  example, if e1= x*(y<0), the value plotted will be x if y<0 and will be 0 otherwise.
  One can save the result of Draw to a histogram:
  ==========
  By default the temporary histogram created is called htemp.
  If varlist contains >>hnew (following the variable(s) name(s), the new histogram created is called hnew and it is kept in the current directory (See the class TDirectory of ROOT).
  Example:
  vnt->Draw("sqrt(x)>>hsqrt","y>0")
  will draw sqrt(x) and save the histogram as "hsqrt" in the current directory.

  By default, the specified histogram is reset. To continue to append data to an existing histogram, use "+" in front of the histogram name;
  vnt->Draw("sqrt(x)>>+hsqrt","y>0")
  will not reset hsqrt, but will continue filling.

- **option** is an option for drawing
  The following options are supported on all types:

"SAME"	Superimpose on previous picture in the same pad
"CYL"	Use Cylindrical coordinates
"POL"	Use Polar coordinates
"SPH"	Use Spherical coordinates

"PSR"	Use PseudoRapidity/Phi coordinates
"LEGO"	Draw a lego plot with hidden line removal
"LEGO1"	Draw a lego plot with hidden surface removal
"LEGO2"	Draw a lego plot using colors to show the cell contents
"SURF"	Draw a surface plot with hidden line removal
"SURF1"	Draw a surface plot with hidden surface removal
"SURF2"	Draw a surface plot using colors to show the cell contents
"SURF3"	same as SURF with in addition a contour view drawn on the top
"SURF4"	Draw a surface using Gouraud shading

- The following options are supported for 1-D types:

"AH"	Draw histogram, but not the axis labels and tick marks
"B"	Bar chart option
"C"	Draw a smooth Curve through the histogram bins
"E"	Draw error bars
"E0"	Draw error bars including bins with o contents
"E1"	Draw error bars with perpendicular lines at the edges
"E2"	Draw error bars with rectangles
"E3"	Draw a fill area through the end points of the vertical error bars
"E4"	Draw a smoothed filled area through the end points of the error bars
"L"	Draw a line through the bin contents
"P"	Draw current marker at each bin
"*H"	Draw histogram with a * at each bin

- The following options are supported for 2-D types:

"ARR"	arrow mode. Shows gradient between adjacent cells
"BOX"	a box is drawn for each cell with surface proportional to contents
"COL"	a box is drawn for each cell with a color scale varying with contents
"COLZ"	same as "COL". In addition the color mapping is also drawn
"CONT"	Draw a contour plot (same as CONT3)

"CONT0"	Draw a contour plot using colors to distinguish contours
"CONT1"	Draw a contour plot using line styles to distinguish contours
"CONT2"	Draw a contour plot using the same line style for all contours
"CONT3"	Draw a contour plot using fill area colors
"FB"	With LEGO or SURFACE, suppress the Front-Box
"BB"	With LEGO or SURFACE, suppress the Back-Box
"SCAT"	Draw a scatter-plot (default)

VIII Dealing with time

In nearly all physics experiments where the time is an important variable, the use of GPS (Global Positioning System) is generalized. This allows to get a microsecond precision clock synchronized throughout the whole world. But the inconvenience is for everyday use that GPS times are expressed as seconds since 1980, January 5, 24:00:00. That means 9 digits numbers. This may be very cumbersome. So there is a need for a convenient way of expressing time while working.

This problem arises in two different fields, when one wants to access data and when one tries to output information related to time.

VIII.1 Time when accessing data

When trying to access data, it would be nice if one could say “I want to access a frame or vector that began 10 minutes ago” or “Give me the vector beginning the 14th of July 2001 at 14:34:05” instead of “time = 632145148”.

VIII.1.1 The reference time

This is why in VEGA, one can access data with respect to a reference time. This reference time is local to a frame channel. To access data, one can use two types of methods. The standard ones, which work in an absolute way, i.e. you have to give the absolute GPS time (like **`VFrameChannel::GetVect`**), and the ones that work by reference to a given time. They differ from the first ones only by the fact that there is an R at the end of the methods name. For instance **`VFrameChannel::GetVectR`**. The arguments are the same except that the time is expressed as the number of seconds since the reference time. It may be negative to define a time that was before the reference one. See the chapter on data access for more information.

VIII.1.2 How to set a reference time

Since each frame channel has its reference time, setting it is done by the VFrameChannel method **`SetRefTime`**. Easy.

```
void SetRefTime(double reftime)
void SetRefTime(char* when, int leapseconds=0)
```

- **`reftime`** is the GPS time expressed as a double to take care of fractions of seconds.

- **when** is a character string that may be one of the following
  - "**now**" for the time at which this setting is done (script executed or command typed at the command line)
  - "**start**" for the beginning time of the first frame indexed by the information database. It's the database start time.
  - a string describing the time (universal time) in the format "**day/month/year(4 digits) hour(1-24):minute:seconds**". For example "**12/04/2000 21:15:34**"
- **leapseconds** is the number of leap seconds that was the reference at the specified time. This is necessary since we are internally using the GPS time. When you express the time as "**12/04/2000 21:15:34**" it will be converted in universal time (UTC), referring to earth's rotation. But GPS is a number of seconds since a reference date. Earth's rotation being non-uniform, these two times drift with respect to each other. The amount of drifting is the number of leap seconds. It is automatically given in a frame header or in a vector. **In case you don't know it, let it to 0**. Be aware that you may have a difference of around 30 seconds (beginning 2000). The number of leap seconds is, as of today, increasing by approximately a little bit more than 1s per year.

Example : If **fc** is a frame channel object opened previously,

```
vega[] fc->SetRefTime("start")
```

will set the reference time to be the start of the info database

```
vega[] fc->SetRefTime("12/04/2000 21:15:34")
```

will set the reference time to the specified date, putting leap seconds to 0. May have a 30 seconds difference with what is expected.

```
vega[] fc->SetRefTime("12/04/2000 21:15:34",32)
```

will set the reference time to the specified date, putting leap seconds to 32. This will be much more correct.

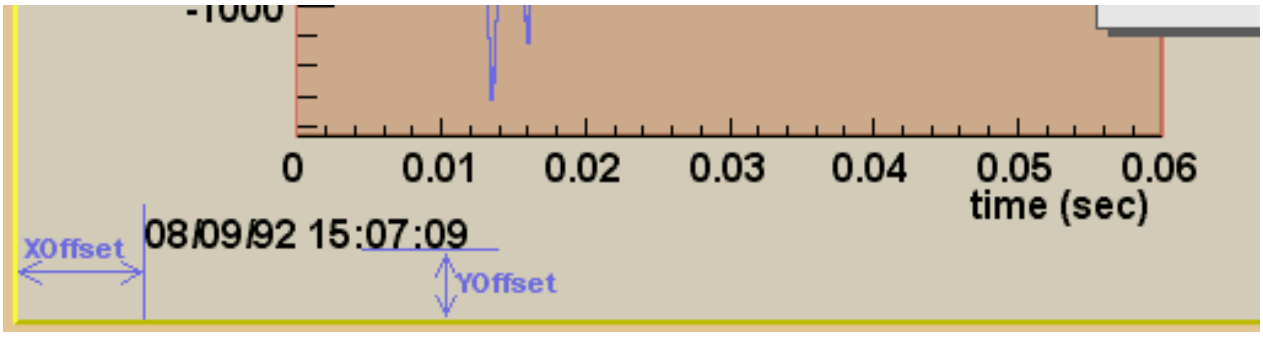
VIII.2 Time when outputting information

VIII.2.1 Plotting frames or vectors start time

From Framelib version v4r0, each FrVect vector has its absolute GPS beginning time included. In VEGA, this start time is automatically plotted below the VSPlot produced when drawing vectors.

Though the default behavior should be all right for most users, one may change the options used for drawing this time by using the global variables gVStyle that holds the style options for the VEGA session. It is described in chapter "The global variables often used".

To change the default behavior, one should set the values in gVStyle BEFORE drawing the vector. These are the x offset, the y offset and the display on/off state, as shown below :



One could use the following methods of gVStyle :

VStyle::SetStartTimeDisplay(true_or_false)

This method toggles the start time display. A call to it :

```
vega[] gVStyle->SetStartTimeDisplay(0)
```

will toggle off all subsequent displays, i.e. no plots will have their start time plotted from then on.

VStyle::SetStartTimeXOffset(float offset)

The position of the time string in the plot is fixed. This method sets the offset of the string with respect to the left side of the pad. A call to it :

```
vega[] gVStyle->SetStartTimeXOffset(0.04)
```

will fix the string position horizontally for all vectors drawn after this call. The default value is 0.02.

VStyle::SetStartTimeYOffset(float offset)

This method sets the offset of the string with respect to the bottom of the pad. A call to it :

```
vega[] gVStyle->SetStartTimeXOffset(0.04)
```

will fix the string position vertically for all vectors drawn after this call. The default value is 0.02.

Once a VSPlot is drawn, similar methods exist to change the position of the start time if needed. They are used in exactly the same way as the VStyle ones, except you need to have a valid pointer to a VSPlot :

VSPlot::SetStartTimeDisplay(true_or_false)

VSPlot::SetStartTimeXOffset(float offset)

VSPlot::SetStartTimeYOffset(float offset)

For instance if vs is a pointer to the last drawn plot, that you get via **gVStyle->SetStartTimeXOffset,** the following :

```
vega[] vs->SetStartTimeXOffset(0.04)
```

will change the x offset of start time.

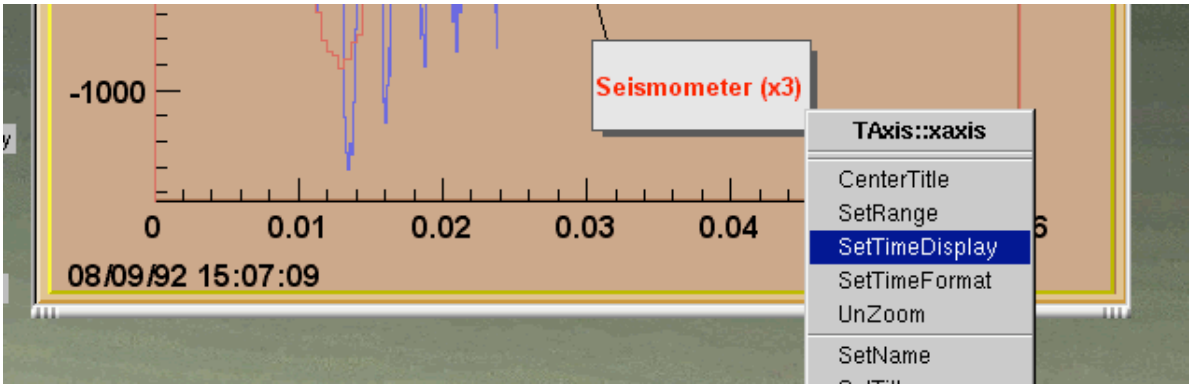
VIII.2.2 Plotting time on time series axis

While it is convenient to plot the start time of the vector, it is also interesting to have the numbers on the axis expressed in seconds or minutes or hours or, why not, days.

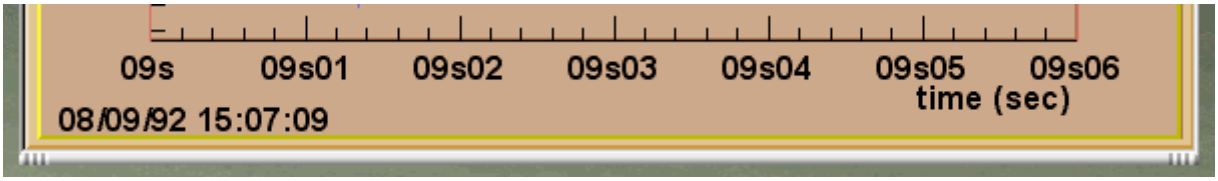
By default, the numbers displayed on an axis are expressed in seconds since the start time of the vector. One can set the axis so that time is expressed in seconds/minutes/hours/days.

VIII.2.2.1 Interactively setting time display on the axis

Interactively, one can set it by using the "SetTimeDisplay" entry of the popup menu of the axis :



The result is the following :



The careful reader will have noticed that the second displayed on the axis is the one of the starting time, also displayed (`08/09/92 15:07:09`). This is general, the time displayed on the axis is the last part of the absolute time, so one can easily and visually connect it with the starting time of the vector. Furthermore, the transitions from one minute to the next or from one hour to the next are correctly handled (we hope so !).

The format of the time display may be set with the SetTimeFormat entry of the popup. This format is the same as the one used in the standard C function strftime. It is a format string that contains :

For the date :

`%a`	`Abbreviated weekday name`
`%b`	`Abbreviated month name`
`%d`	`Day of the month (01-31)`

%m	Month (01-12)
%y	Year without century

For time :

%H	Hour (24 hour clock)
%I	Hour (12 hour clock)
%p	Local equivalent of AM or PM
%M	Minute (00-59)
%S	Seconds (00-61)
%%	%

For example the standard string used in Europe is "`%d/%m/%y %H:%M:%S`".

VIII.2.2.2 Set axis time parameters in a script

In a script, one will need to set/unset/change the time format or parameters with statements. There are two ways.

VIII.2.2.2.1Directly

The first is to access the axis object of the plot. It is of type TAxis and, given a plot "vs" is accessed via "vs->GetXaxis()" or "vs->GetYaxis()".

Suppose we have a vector ve of type FrVect* already extracted. A typical use would be :

Draw the vector and get the associated VSPplot :

```
vega[] gVM->Draw(vect)
vega[] vs = gVM->GetLastPlot()
```

Set the time display on the X-axis. In order to do this, since time display are axis characteristics, one first accesses the axis object with **`vs->GetXaxis()`** and then calls the **`SetTimeXXX`** method :

```
vega[] vs->GetXaxis()->SetTimeDisplay()
```

Set the time format :

```
vega[] vs->GetXaxis()->SetTimeFormat("a user format")
```

of course, replace "a user format" by what you want to be the format. Format options are described in the previous paragraph.

VIII.2.2.2.2Through the style

The second possible way is to use the gVStyle variable, so that you set it once and all the subsequent plots will be drawn with the specified options, as for example, time on axis turned on, as in :

```
vega[] gVStyle->SetTimeOnXAxis(1)
```

The other available option are :

```
gVStyle->SetTimeFormat("a user format")
```

Again, see preceding paragraphs for the format.

Remember : use the style global gVStyle BEFORE drawing the plot.

IX The Data Display as a data viewer

IX.1 Goals

The Data Display is commonly used inside the VIRGO collaboration for interactive viewing of frames online (via the network) or offline (in files). The display is done inside a ROOT canvas, so it was natural to incorporate the Data Display functionalities inside VEGA. A user is then able to view some data, then process the graphical result via a script.

For a detailed explanation of the Data Display use, the reader is invited to look at the Data Display user's manual. We will focus here on the interaction with VEGA.

IX.2 The Data Display inside VEGA

Displaying data interactively using the Data Display may be done inside VEGA in two ways:

- use the command `dataDisplay()`:
  `vega[] dataDisplay()`
  This will launch the usual Data Display
- launch the data display on a frame channel previously opened.

IX.2.1 Displaying data from a frame channel source

The frame channel is the main data providing mean in the VEGA "philosophy". It is thus natural to be able to view data coming from a frame channel inside the Data Display. If "fc" is an already opened frame channel, this is done by the command:

```
vega[] fc->StartViewer()
```

The Data Display will be opened with the main browser window showing the channels inside frames from the frame channel.

IX.2.2 Interaction with the produced plots

Once the user has made the desired plots, he is able to interact with these plots from inside VEGA. A global variable named gDataViewer represents the data display. One can interact with it with the following methods :

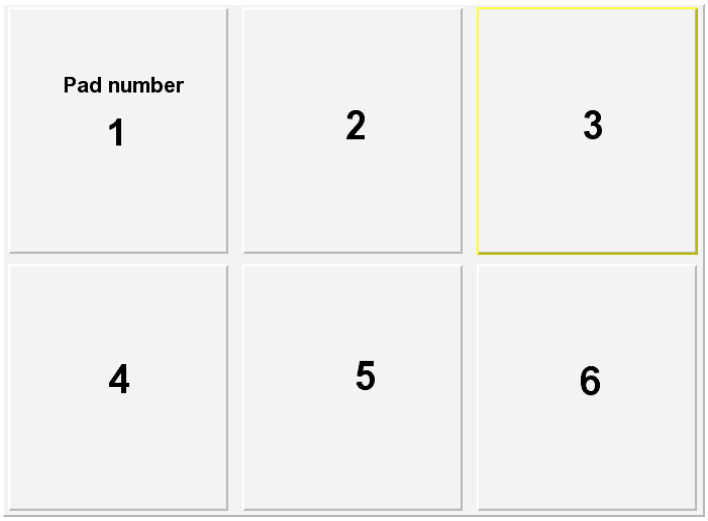
gDataViewer->GetPlot(i)

will return a TObject* corresponding to the plot in the pad n° i. This object should be

casted to a TH1F* or a TH2F* depending on the type of the plot in the corresponding pad, like in :

```
vega[] h1 = (TH1F*)gDataViewer->GetPlot(3)
```

Reminder : the pads are numbered from up to down and left to right, like in :



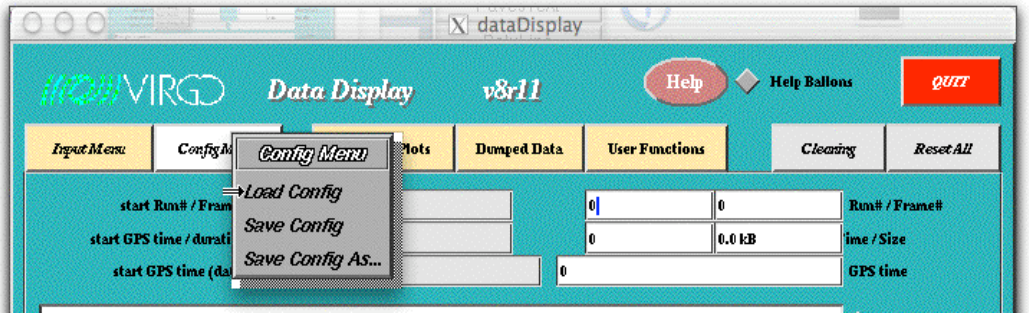
The canvas created by the data display is called gDDCanvas and is accessible globally after its creation. One can for example put the y scale of the pad number 2 in log with the commands:

```
vega[] gDDCanvas->cd(2)
vega[] gPad->SetLogy(1)
```

Reminder : gPad always points to the current pad.

IX.2.3 Using configuration files produced by the standalone dataDisplay application

The standalone Data Display application produces configuration files ("cards files") that a user may wish to reuse. This may be done if a cards file points to files (not to online Cm producer) by starting the data display interactively with the `dataDisplay()` command and then loading the cards file from the Config Menu :



X Signal processing and algorithms

The signal processing functions present in VEGA rely on the Frv analysis library developped in VIRGO. A default version of the library is provided with the VEGA sources but may be upgraded at any time.

The API of this library are not yet finalized but come very close to this point.

This library uses the FFTW (http://www.fftw.org) library for computing fft's. The installation of FFTW is not difficult and described in the installation documents, i.e. INSTALL file in the distribution or "Installation of ROOT"

The sources and documentation of Frv may be found http://wwwlapp.in2p3.fr/virgo/FrameL/. We will just give an example to demonstrate the use of the library to make an FFT of a vector.

A structure FrvRFFT that holds the result of an fft transform of a frame vector FrVect is defined. The algorithm itself is build and called with FrvRFFTFor (0,frvect) as for example in (suppose vect is an FrVect) :

```
vega[] fft = FrvRFFTFor (0,vect)
```

This will create a standard fft and apply it to the vector vect. On outputing, the fft structure contains the result of the processing in the form of FrVect's that one can plot as usual :

```
vega[] gVM->Draw(fft->amplitude)
```

For the amplitude part of the fft. Once the fft object is not needed anymore, the user should delete it with

```
vega[] FrvRFFTFree(fft)
```

Look at the example scrollspectrum.C in the tutorials.

More information will be added as soon as the Frv library gets a standard shape.

XI The global variables often used

XI.1 The active manager gVM

XI.2 The active style gVStyle

XII Examples of macros

The examples are available interactively in the VEGA site : tutorials. We suppose you have created the demo frame files by executing the macros "create_testfr.C" and the corresponding local metadatabase with "create_DB.C" (this is just a two lines script that does what is described above for creating a database).

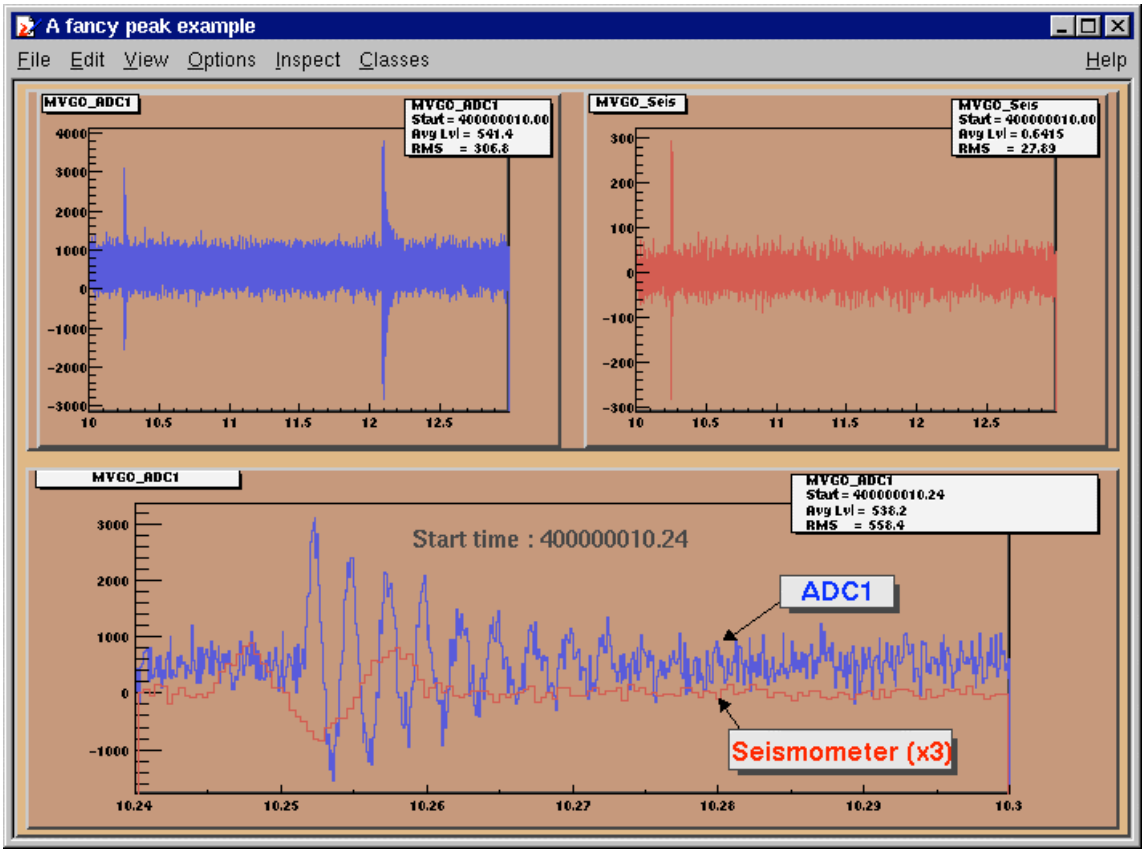
XII.1 Example 1 : Displaying a peak

The first example is showing how one can display a potentially interesting peak, zooming in and plotting two channels on top of each other (macro **peak.C**) :

```
{
// This example shows how to plot a region of interest.
// To see the result of this macro, click  here
// The access of data is made by its time.
// A much fancier example is given in  here

   gROOT->Reset();

   // Open a new window (canvas)

   c1 = new TCanvas("c1","A peak example",1);

// ===== Creates the needed pads by dividing the initial canvas =====

   c1->Divide(2,2);

// ===== Opens the frame channel =====

   vd = new VFrameChannel("demoDB.root");

// ===== Gets the needed vectors and draws them =====
// ===== First the upper plots                  =====

   FrVect* frv1 = vd->GetVect("adc.MYGO_ADC1",400000010.,3.);
   FrVect* frvSeis = vd->GetVect("adc.MYGO_Seis",400000010.,3.);

   c1->cd(1);                              // select the first pad
   gVM->Draw(frv1);

   c1->cd(2);                              // select the second pad
   gVM->Draw(frvSeis);
// ===== Gets the needed vectors and draws them =====
// ===== Second the same plots superimposed     =====
```

```
    // Gets the needed vectors again but zooming on the region of interest
    frv2 = vd->GetVect("adc.MYGO_ADC1",400000010.24,0.06);
    frvSeis2 = vd->GetVect("adc.MYGO_Seis",400000010.24,0.06);

    // Draws them in the second pad, superimposed on top of each other
    c1->cd(3);
    gVM->Draw(frv2);
    gVM->Draw(frvSeis2,"same");
}
```

To show the result when displaying a the same peak but in a very fancy way (well, this is all a matter of taste...), with the macro "peak2.C", here is the result :



XII.2 Example 2 : scrolling data

This example shows how to display a channel extracted from a database with scrolling one second at a time. The channel is extracted in the form of FrVect vectors. Each extracted FrVect does not match the length of the frame it belongs to. In fact, the user should not care about that.

The example is in the file **scroll.C**

```
{
// This simple example opens a demo local database and plots a vector.
// The vector is scrolled 1 s at a time, nvecmax times.
// The extraction of the vector shows that one can select a vector
// without having to think about the frame boundaries.
// The length of the vector is arbitrary and is only limited by the available
// amount of memory.

   gROOT->Reset();

   FrVect* frv;
   Double_t time;

// Max number of vectors displayed
   Int_t nvecmax = 20;
// Time step of the scroll
   Double_t tistep = 1;
// Length of the displayed vector
   Double_t veclength = 4;

// Open a new window (canvas)
   c1 = new TCanvas("c1","c1",1);

// Open a frame channel
   vd = new VFrameChannel("demoDB.root");

// Start time
   time = 400000000.1;

// Get the first vector
   frv = vd->GetVect("adc.MYGO_ADC1",time,veclength);
   Int_t nvec = 0;
   while ( (frv!=0) && (nvec<nvecmax) ) {
      nvec++;
      printf("nvect = %d, time = %fn",nvec,time);

// Draws the vector
      gVM->Draw(frv);
// Update the canvas
      gPad->Update();

      FrVectFree(frv);

      time += tistep;
      frv = vd->GetVect("adc.MYGO_ADC1",time,veclength); // Get the next vector
   }
}
```

XII.3 Example 3 : histogramming some data and fitting the result

This example shows how to use histogramming with the VManager. Access is made to the data and an interesting part containing a peak is displayed, as well as a histogram of the data. The histogramed data is accumulated over several frames and, at the last frame, a fit is made to the histogram. One fits a gaussian to show the deviation from gaussianity.

The example is in the macro **scroll2.C**

```
{
// This example plots a vector and accumulates a histogram of the values
// of this vector.
// Only the part of the vector that corresponds to the interesting part
// are plotted (a peak at the beginning of the frames)
// The histogram is accumulated for nframmax frames
// To see the result of this macro, click  here

            gROOT->Reset();

            FrameH* fr;
            Double_t time;

// Max number of frames displayed
            Int_t nframmax = 14;

// Open a new window (canvas) and divide it by 2
            c1 = new TCanvas("c1","c1",1);
            c1->Divide(2,1);
            c1_1->SetFillColor(44);
            c1_1->SetFrameFillColor(41);
            c1_2->SetFillColor(44);
            c1_2->SetFrameFillColor(41);
            c1_2->SetLogy();
            c1_2->SetGridx();
            c1_2->SetGridy();

// Open a frame channel
            vd = new VFrameChannel("demoDB.root");

// Start time
            time = 400000000.1;

// Histogram to constrain the limits
            hisfr = new TH1F("hisfr","Histogram of MYGO_Seis
                                      values",100,0.,3000.);
            hisfr->SetMarkerColor(4);
            hisfr->SetMarkerStyle(8);
```

```
            hisfr->SetMarkerSize(0.7);

// Get the first frame
            fr = vd->GetFrame(time);
            Int_t nfram = 0;
            while ( (fr!=0) && (nfram<nframmax) ) {
                  nfram++;
                  printf("nfram = %dn",nfram);

//    Draws the vector (only in the signal region)
                  c1->cd(1);
                  gVM->Draw(fr,"adc.MYGO_ADC1",0.13,0.07);
//    Get the corresponding VSPlot and set the color to blue
                  VSPlot* vfp = gVM->GetLastPlot();
                  vfp->SetLineColor(4);

//    Fill the histogram (accumulates it) and draw it
//    the + option in front of the name of the histogram
//    means that it's content is not erased before each fill
                  c1->cd(2);
                  gVM-
                                          >DrawHist(fr,"adc.MYGO_ADC1",0.13,
                                          0.07,"+hisfr","e1");

//    Update the canvas
                  gPad->Update();

                  FrameFree(fr);

                  fr = vd->GetNextFrame(); // Get the next frame
            }

// Fits the histogram and draws the fit result
            hisfr->Fit("gaus");
            hisfr->SetMarkerColor(4);
            hisfr->SetMarkerStyle(8);
            hisfr->SetMarkerSize(0.7);
            hisfr->Draw("e1");
}
```

XII.4 Example 4 : Extracting some slow monitoring data and doing some simple plots

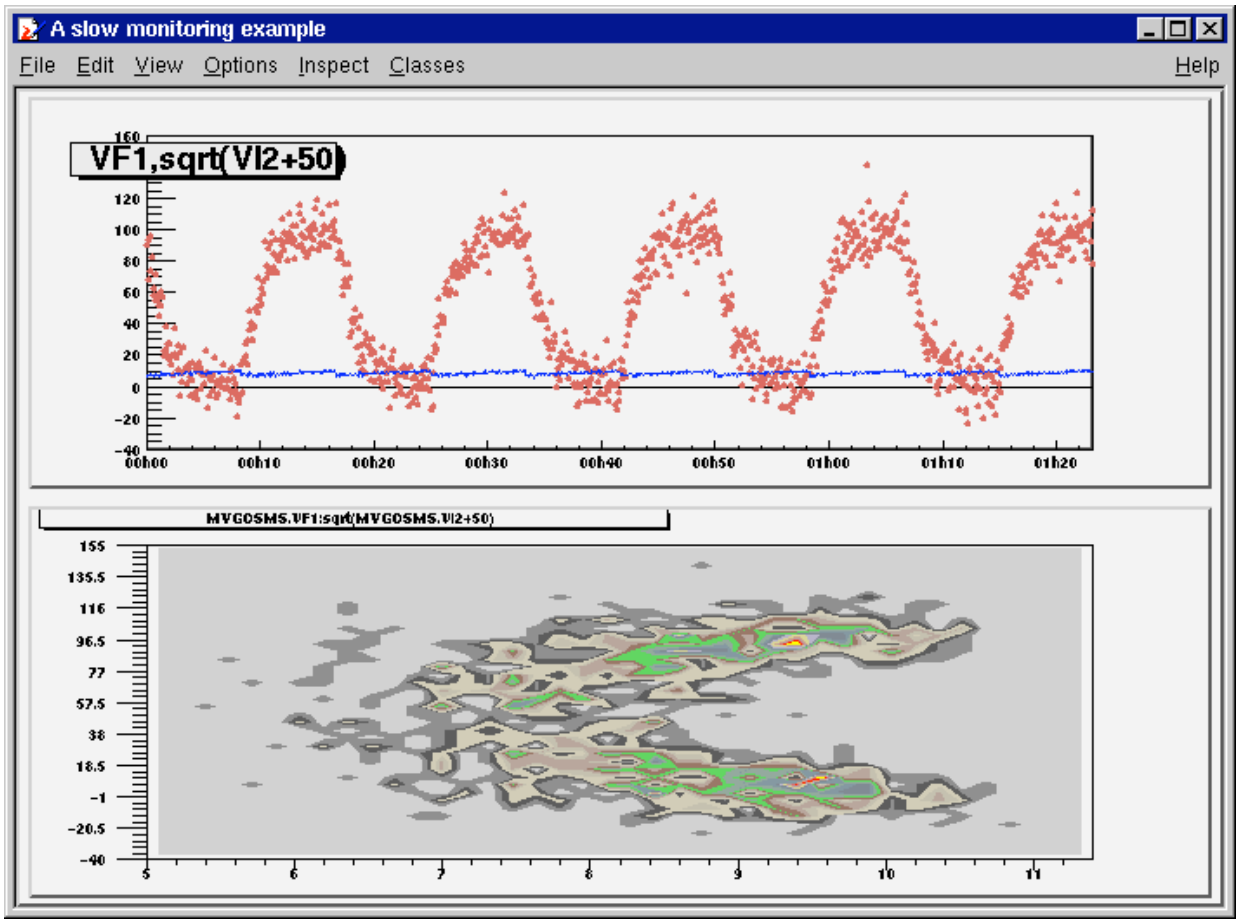
This example shows how to extract slow monitoring data and plot variables or combinations of them.

The example is in the macro **sms.C**

```
{
// This example shows how to plot a set of slow monitoring values.
// To see the result of this macro, click begin_html <a
href="gif/sms.gif">here</a> end_html
// A ntuple is build and used to plot various combinations of variables.

   gROOT->Reset();

// Open a new window (canvas)

   c1 = new TCanvas("c1","A slow monitoring example",1);

// ===== Creates the needed pads by dividing the initial canvas =====

   c1->Divide(1,2);

// ===== Opens the frame channel =====

   vdsms = new VFrameChannel("SMSdemoDB.root");

// ===== First builds the ntuple from the MYGOSMS station values =====

   nt = vdsms->ExtractSMS("nt","MYGOSMS",400000000,5000);

// ===== Then draws the plots =====

   c1->cd(1);                          // select the first pad

  // Sets some style options
   gStyle->SetMarkerColor(50);
   gStyle->SetMarkerStyle(8);
   gStyle->SetMarkerSize(0.5);

  // Draws the value VF1 with respect to time
  // t is automatically filled as the first variable of the ntuple at
extraction
   nt->DrawSeries("MYGOSMS.VF1","","P");
   vs = gVM->GetLastPlot();
   vs->SetTitle("");

  // Sets some style options
   gStyle->SetLineColor(4);

  // Superimposes the values of sqrt(VI2+50) (it's an example)
   nt->DrawSeries("sqrt(MYGOSMS.VI2+50)","","same");
```

```
   // Draws a title for the graphs
    pl = new TPaveLabel(-400,135,1000,156,"VF1,sqrt(VI2+50)","br");
    pl->Draw();

// ===== In the second pad, draws a histogram =====
// ===== of values VF1 with respect to sqrt(VI2+50)    =====

    c1->cd(2);                              // select the second pad
   // Draws the histogram
    nt->Draw("MYGOSMS.VF1:sqrt(MYGOSMS.VI2+50)","","cont0");
}
```

Here is the result of this macro :



XII.5 Example 5 : Extracting a vector and plotting spectra

This example shows how to use the fft algorithms to plot spectra.

The example is in the macro **scrollspectrum.C**

```
{
// This simple example opens a demo metadatabase,
// plots a vector and it's spectrum.
// The vector is scrolled 1 s at a time, nvecmax times.
// The extraction of the vector shows that one can select a vector
// without having to think about the frame boundaries.
// The length of the vector is arbitrary and is only limited by the
available
// amount of memory.
// It is also shown how to use the FFT from the signal analysis library

   gROOT->Reset();
   gBenchmark->Start("scrollspectrum");


   FrVect* frv;
//   Max number of vectors displayed
   Int_t nvecmax = 20;
//   Time step of the scroll
   Double_t tistep = 1;
//   Length of the displayed vector
   Double_t veclength = 4;
//   Start time
   Double_t time = 400000000.1;
// Open a new window (canvas) and split it
   c1 = new TCanvas("c1","c1",1);
   c1->Divide(1,2);
   c1->cd(2);
// Open a frame channel
   vd = new VFrameChannel("demoDB.root");
// Get the first vector
   frv = vd->GetVect("adc.MYGO_ADC1",time,veclength);
   Int_t nvec = 0;
   while ( (frv!=0) && (nvec<nvecmax) ) {
      nvec++;
      printf("nvect = %d, time = %f\n",nvec,time);
// Draws the vector in the first pad
      c1->cd(1);      // go to the first pad
```

```
      gVM->Draw(frv);
// Calculates the spectrum
      FrvRFFT* fft;
      fft = FrvRFFTFor(0,frv);

// Draws the spectrum in the second pad
      c1->cd(2);      // go to the second pad
      gVM->Draw(fft->amplitude);
      gPad->SetLogx(1);
      gPad->SetLogy(1);
// Update the canvas
      c1->cd();
      gPad->Update();
// Release the various variables allocated
      FrVectFree(frv);
      FrvRFFTFree(fft);


      time += tistep;
      frv = vd->GetVect("adc.MYGO_ADC1",time,veclength); // Get the next
    vector
   }


   gBenchmark->Show("scrollspectrum");
// Do some garbage collection
   if (frv) FrVectFree(frv);
   delete vd;
}
```

XIII How to compile your own code for VEGA or use the VEGA libraries in your code

XIII.1 Loading an existing shared library

If a shared library that was suitably prepared (we will see what that means) exists already, it is possible to extend VEGA by loading it. The command to do that is ".L" as in

```
vega[] .L scrollspectrum.so
```

Once this has been done, the functions and methods that are defined in the library are available at your fingertips in the interpreter.

What means "suitably prepared" is that one has to build the stub functions that make the interface between the interpreter and the real functions. This is however an automatic process that is using a tool called rootcint. We will describe it more extensively later.

XIII.2 Compiling a script and making a shared library of it

The script compiler was introduced in the chapter " How to compile a macro?". Before going further, let's remind what was said.

Sometimes (in fact quite often) a script may become so big or be so processing-hungry that one would like to have it compiled and not interpreted. There is included into ROOT what is called a script compiler that allows compiling very easily the scripts you are going to produce.

It starts by calling the C++ compiler that was used to build ROOT and builds a shared library. This shared library is then loaded and the function defined in the script is executed.

The shared library that is produced is going to stay and you will be able to extend VEGA by loading this library later.

But before using the script compiler, one has to prepare a little bit his script, though it is quite simple.

XIII.2.1 Preparation of the script

You have to

1) Do not use extensions to C/C++ provided by the CINT interpreter. That means for example that while in the interpreter you do not need to declare variables whose type is known from the context, this is mandatory if you want to be able to compile. Since the compiler is not supposed to know the context of the interpreter. An example. In the interpreter, you can use statements like :

```
c1 = new TCanvas("c1", "Test",1);
```

where you do not declare the variable c1, you should do instead :

```
TCanvas* c1 = new TCanvas("c1", "Test",1);
```

More generally, use very standard C/C++ and declare all your variables

2) You should add the includes you use at the beginning of the file. Once again, this is for the real compiler that will do the work. Usually, the headers to be added are composed of the name of the classes used followed by the .h suffix. In case of included C code, the most commonly used are `"FrameL.h"` for all the Framelib calls, and the standard headers of the Virgo standard signal library.

3) In case you want to execute immediately the first function of the script you compile, you have to name the function in your script with the same name as the file it is contained in. For example the file my_test.C should begin with

```
my_test()
{
    your code….
}
```

The tutorial script called `scrollspectrum.C` has been prepared this way, check it.

XIII.2.2 Compiling the script

What remains to be done is to compile the script. If we want to execute the script called `scrollspectrum.C`, we just have to do (once in the `vegatutorial` directory):

```
vega[] .x scrollspectrum.C++
```

notice the ++ at the end of the command. This tells the interpreter to compile `scrollspectrum.C`, build a shared library called `scrollspectrum.so`, load this shared lib and execute `scrollspectrum()`.

The resulting library scrollspectrum.so stays here and you can reuse it, that is reload it at a later time, with

```
vega[] .L scrollspectrum.so
```

furthermore, the scrollspectrum() function is available in the interpreter and may be used in some other script!

All this also works with loading the script after compiling:

```
vega[] .L scrollspectrum.C++
```

is a valid statement. This will build the library and load it as before. It will just not execute any function.

XIII.2.3 Limitations

The script compiler is not intended to be used for big projects. Do not expect to compile libraries that are made of many subroutines in many different source files. It is quite efficient though for checking that a code will compile before effectively using it.

It is also possible to overcome some CINT limitations using the script compiler since it uses the full-featured compiler you are used to have.

XIII.3 Building a standalone executable

XIII.3.1 Building an executable that is linked with the ROOT and VEGA libraries

The directory "test" contains a few examples of programs that use the VEGA and ROOT classes. The "`main_metadb.cc`" and "`main-display.cc`" programs are just skeletons that may be used as a starting point to build bigger programs.

The main difficulty for their compilation is to put all the necessary includes and to link with all the necessary libraries. Furthermore, one has to use the same compiler as the one used for compiling ROOT and VEGA. This ensures the less possible problems.

In order to simplify as much as possible the determination of the various includes, libraries or parameters to use, a utility script is provided, called "`vega-config`" that shows them. This utility is generated at VEGA compile time and returns all the parameters used for compiling VEGA.

For example, to determine what C++ compiler was used, do:

```
vega[] vega-config --cxx
g++
```

XIII.3.2 Options of the vega-config utility

The options available for "`vega-config`" are the following:

--help	: prints a help message
--metadb-set	: only output parameters (includes, libraries) relevant to metadatabase for the following options
--display-set	: only output parameters (includes, libraries) relevant to display for the following options
--signal-set	: only output parameters (includes, libraries) relevant to signal for

the following options

an OR is made of the three previous flags

--cxx	: command to invoke the C++ compiler
--cc	: command to invoke the C compiler
--ld	: command to invoke the linker
--prefix	: gives the prefix path used for root
--cflags	: outputs the flags used by the C/C++ compiler to compile root and VEGA
--ldflags	: flags used when invoking the linker
--soflags	: flags invoked for creation of a shared object
--coptflags	: outputs the optimisation flag used by the C/C++ compiler to compile root and VEGA
--includes	: output include options used when compiling an object for VEGA, following the previous flags
--nonew	: do not include libNew for the following options
--libs	: output libraries options used when compiling an object for VEGA, following the previous flags
--glibs	: output libraries used by VEGA or ROOT when making an application using the ROOT GUI
--defines	: output defines that were used for compiling VEGA following the previous flags
--version	: gives the VEGA version

The flags may be combined to give a complete command and characters that do not represent any flag are output as is. Examples :

```
> vega-config --cxx -c toto.cc

g++ -c toto.cc
```

One can build a complete command to compile a source code :

```
> vega-config  --cxx --cflags --includes -c toto.cc

g++ -fPIC –fsigned-char –I/home/toto/vega/src/include
–I/virgoApp/Fr/v4r25/src –I/home/toto/root/include
–I/home/toto/fftw/include –c toto.cc
```

One clearly sees that the result may not be simple, and that it depends on the particular configuration of the system used. The vega-config command and flags are the same however and insure that one is able to compile his program the same way whatever the system.

The vega-config script only outputs the command, but does not execute it. In most shells, putting the command between back quotes will execute the command and then the output of the command :

```
> `vega-config  --cxx --cflags --includes -c toto.cc`
```

Now, it is possible to build relatively easily any program. First try with the ones provided in the test directory.

XIII.3.3 Using only a subset of capabilities

To allow saving of resources, one may be willing to use only relevant parts of the VEGA/ROOT system. For example use only metadatabase, without graphics and display. There are three flags of the vega-config utility that restrict the output to only relevant parts of a particular set :

--metadb-set : only output parameters (includes, libraries) relevant to metadatabase for the following options

--display-set : only output parameters (includes, libraries) relevant to display for the following options

--signal-set : only output parameters (includes, libraries) relevant to signal for the following options

Try yourself the following commands and see the difference :

```
> vega-config --includes –libs
> vega-config –metadb-set --includes --libs
```

XIII.4 Building a general shared library usable with VEGA

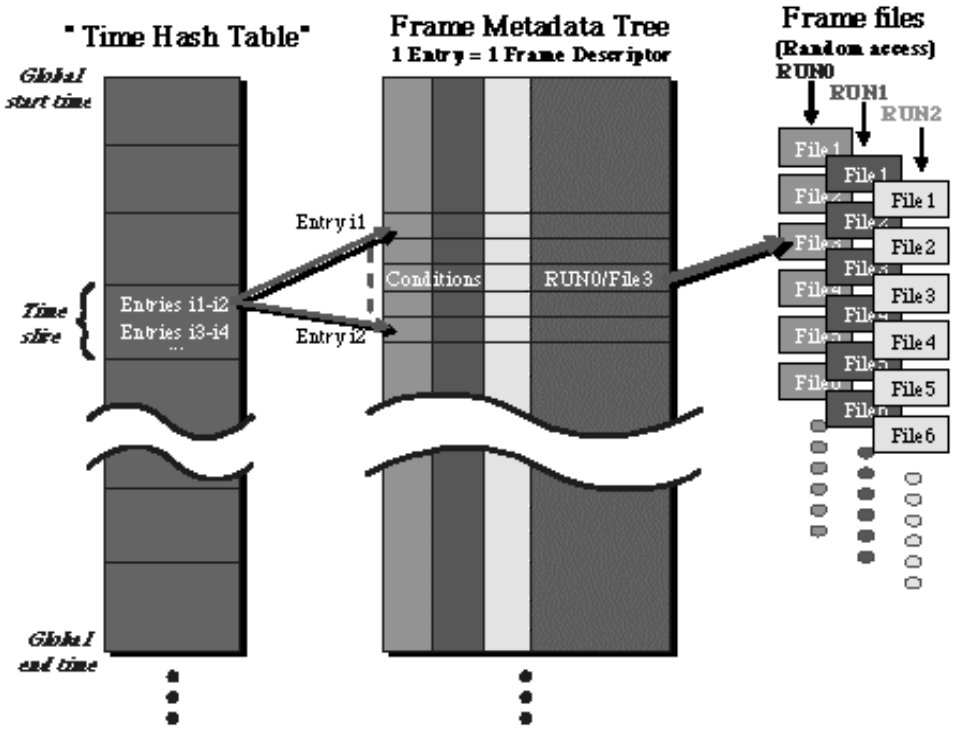
I Appendix A : Local metadatabase structure and principles

I.1 Metadatabase for easy data access

I.1.1 Principle and structure

From the user's point of view, the analysis environment should allow her/him to access simply any vector or frame that is contained in a set of frame files, even remotely. One should also be able to access vector not caring about frame boundaries. It should also be possible to make accesses conditioned by some trigger, slow monitoring value or user-defined condition. Furthermore, one should be able to manage a set of files that is as big as possible (of the order of 1 TB or more)

The option chosen in VEGA to solve this problem is to build a database that contains metadata about frames and indexes these frames. The structure of the database is drawn on the following figure :



The complete frame information is kept in the frame files, while the database serves as an index to access them. The access is a multistep process. First, one gives the starting time of the vector one wants to a frame channel. This starting time is given to the metadatabase and is used to determine, with the help of what is called a "time hash table", the frame files that possibly contain the desired information. Then, only the metadata corresponding to these files are searched. The corresponding metadata is sent back to the frame channel that will access the desired frame in the relevant file or reconstruct the desired vector and give it to the user.

The metadata is kept in a container, called Tree, introduced in ROOT and specifically designed to handle big amounts of data and access it quickly. The tree structure can also contain conditions for frame access.

I.1.2 Creation of a metadatabase

```
VFrameMetaDB(char* filename, char* mode, char* framefilenames, char*
        opt)
```

To create a local database, we need to create a database object of class VFrameMetaDB. At the same time, we can build it, looking for frame files. As, for example, in

```
vega[] vd = new VFrameMetaDB("demoDB.root","CREATE","./")
```

we call the VFrameMetaDB constructor.

The first parameter is the name of the database file.

The second one is the mode with which we open the database. Here, it is opened in "CREATE" mode, since we want to build it.

the third parameter is the path to the directory containing the frame files. Here, it is "./", meaning the local directory. You can put whatever path you like and specify particular files/directories with wildcards.

The search is by default recursive, all the contained directories will be searched recursively. If you want to turn this option off, add a last parameter and specify "S"

That's it ! The amount of time needed to index the files depends on the number and size of those files. But typically, it consists of a sequential read of all the files. This shouldn't be much slower than the speed of the connection to the disk containing the directories.

I.1.3 Access to metadata

I.1.3.1 Opening an existing database

Opening an existing metadatabase may be done by using the constructor in "READ" mode, which is the default :

```
vega[] vd = new VFrameMetaDB("./demoDB.root")
```

One may replace "./demoDB.root" with the path/name of his local database.

I.1.3.2 Extracting metadata

Metadata are extracted from the database with

```
void GetMetaData(VMetaData* meta, Double_t time)
```

This method extracts a metadata at an absolute time.

As in

```
vega[]vd->GetMetaData(meta, time)
```

where time is a double expressing a time that is contained in the frame and meta is a pointer to a VMetaData metadatabase object, created by the user, to be filled. This method fills the meta object with the right information. In this way, the user always keeps control of creation/deletion of the metadata objects.

Once one has extracted one metadata, it is possible to extract the next one or the preceding one in the database :

```
GetNextMeta(VMetaData* meta)
```

For example :

```
vega[] vd->GetNextMeta(meta)
```

or

```
GetPreviousMeta(VMetaData* meta)
```

For example :

```
vega[] vd->GetPreviousMeta(meta)
```

where meta is a pointer to a VMetaData metadatabase object, created by the user, to be filled. This method fills the meta object with the right information. In this way, the user always keeps control of creation/deletion of the metadata objects.

Be careful, while GetFrame is checking to see if the requested time is really contained in one of the frames of the database, GetNextMeta and GetPreviousMeta do not. They will simply return you the metadata that is the next (or previous) one in time in the database, even if it's years away.

I.1.4 Getting general information about the metadatabase and it's contents

I.1.4.1 Getting the start time of the metadatabase

To get the start time of the first frame indexed in the metadatabase, one has to use the VFrameMetaDB method

```
double GetStart()
```

For example if vd is a valid metadatabase,

```
vega[] vd->GetStart()
```

will output this start time while

```
vega[] st = vd->GetStart()
```

will put it in a double that may be reused.

I.1.4.2 Printing information about the metadatabase

One can get some information about the metadatabase by using the VFrameMetaDB method

```
double Print()
```

For example if vd is a valid metadatabase,

```
vega[] vd->Print()
```

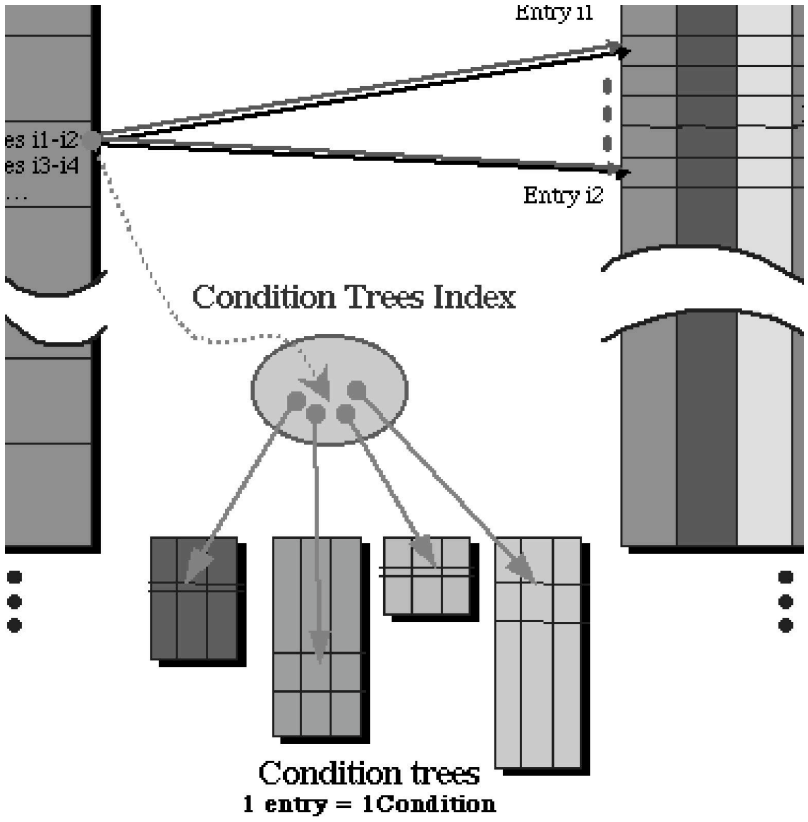
This will show a rather extensive output of the metadatabase content.

I.2 Dealing with selected or triggered data

I.2.1 Condition information in the metadatabase

When the metadatabase is build by reading the frame files, the trigger information contained in each frame is retrieved and arranged in such a way to allow easy extraction of frames or vectors satisfying some selection.

The triggers (structures of type FrTrigData in the FrameLib) are converted to more general objects called conditions. This will allow to use in the future as conditions other information such as slow monitoring data or quality information. The user may even add his own conditions, without copying the whole files just to add a FrTrigData structure. A simple scheme of the database was given above. This scheme is now enhanced by the addition of condition trees and an index for fast access:



I.2.2 Extracting metadata with a condition

There are two ways of extracting metadata that satisfy a given condition: directly or sequentially. The direct method will be used if a particular frame, which approximate time is known, is to be extracted. The sequential method is used if one needs to process or view sequentially all the frames that correspond to a given selection expression.

I.2.2.1 Direct methods

The method of VFrameMetaDB that allow a direct conditioned access to metadata is:

GetNextFrame(VMetaData* meta, Double_t time, char* selection)

This method extracts a metadata at an absolute time. "selection" is a selection expression referring to conditions existing in the database such as "Trig1.amp>50 && Trig2.amp>2".

Example

```
vega[]meta = vd->GetNextMeta(time,"Trig1.amp>50&&Trig2.amp>2")
```

the search will begin from time "time". The metadata "meta" will be filled by information from the one containing the start time of the first interval satisfying the selection expression. This method fills the meta object with the right information. In this way, the user always keeps control of creation/deletion of the metadata objects.

I.2.2.2 Sequential methods

In order to access sequentially all the metadata of interest, satisfying a selection expression, one has to have an object that will point to the intervals of interest. So came the idea of condition sets. Once a condition set has been defined, one can use it to extract metadata that are recorded at the corresponding time. The methods of VFrameMetaDB to do so are:

GetNextMeta(VMetaData* meta, VConditionSet* condset)

This method extracts a metadata at an absolute time. "condset" is a condition set defined as explained in the paragraph "Condition Sets"

Example

```
vega[] vd->GetNextMeta(meta, condset)
```

The search will be governed by the condition set "condset". The metadata returned will be the one containing the start time of the next interval pointed to by "condset".

These methods have to be called sequentially in a loop.

This method fills the meta object with the right information. In this way, the user always keeps control of creation/deletion of the metadata objects.

Table of Contents

I Introduction 1
- I.1 What is ROOT ? 1
- I.2 Objectives of this document 2
- I.3 Installation of VEGA/ROOT 2
- I.4 How to find information ? 2

II An interactive session 3
- II.1 Basic intrinsic operations 3
- II.2 The interpreter CINT, graphical interaction through a few examples 3
  - II.2.1 First very simple example 3
  - II.2.2 Graphical output example 4
  - II.2.3 Classes, methods and constructors 5
  - II.2.4 ROOT related specifics 6
  - II.2.5 Graphical output example: User interaction 6
  - II.2.6 Second example : Building a multi-pad canvas 8
  - II.2.7 Inheritance and data encapsulation 9
  - II.2.8 ROOT related specifics 10
- II.3 An example of a macro 10
  - II.3.1 C++ notions used: method overloading 13
- II.4 How to compile a macro ? 14
  - II.4.1 Preparation of the script 14
  - II.4.2 Compiling the script 15
  - II.4.3 Compiling a more complex code 15

III Hands on VEGA 16
- III.1 Preparation 16
- III.2 Opening a Frame Channel 16
- III.3 Extracting a frame using a frame channel and plotting it 17
- III.4 Extracting a vector using a frame channel and plotting it 18
- III.5 Extracting a 2D vector (image…) from the frame channel and plotting it 19
- III.6 Building a metadatabase for more complex tasks 19
- III.7 Extracting a frame with a condition or selection 20
- III.8 How to deal with slow monitoring data ? 21

IV More on C++ (for the interested reader) 25
- IV.1 C++ notions used: classes, methods and constructors 25
- IV.2 C++ notions used: inheritance and data encapsulation 26
  - IV.2.1 Inheritance 26
  - IV.2.2 Method overriding 27
  - IV.2.3 Public and private : data encapsulation 27
- IV.3 C++ notions used: operators new and delete 29

V Accessing and managing Gravitational Waves data 31
V.1 Accessing frames through the Framelib.....31
V.2 General scheme for data access.....31
V.3 Different information databases for different uses 32
V.3.1 Simple access: direct files.....32
V.3.2 More complex case: metadatabase 33
V.3.3 Full case: bookkeeping database 33
V.4 Access to the data through a Frame Channel 33
V.4.1 Opening a frame channel and connecting to an information database.....33
V.4.2 Accessing data through a Frame Channel 34
V.4.3 Getting general information about the information database and it's contents 36
V.5 Dealing with selected or triggered data 37
V.5.1 Condition information in the information database 37
V.5.2 Condition Sets 37
V.5.3 Extracting frames with a condition.....39
V.5.4 Extracting vectors with a condition 40
V.5.5 Getting information about condition sets 42
V.6 N-tuples adapted for GW data analysis.....43
V.6.1 Building VNtuple.....44
V.6.2 Filling an N-tuple and getting data.....45
V.6.3 Drawing 46

VI Analysis of data coming from a frame channel: the selector.....47

VII Representing Gravitational Waves data 49
VII.1 Series plots 49
VII.2 Graphical managers of frames.....49
VII.2.1 Definition of managers 49
VII.2.2 Building a manager : the global gVM.....50
VII.2.3 General use.....50
VII.2.4 Drawing methods.....50
VII.2.5 Change of the plots attributes 59
VII.2.6 Management of the plots produced.....60
VII.3 Representing slow monitoring data 60

VIII Dealing with time 66
VIII.1 Time when accessing data.....66
VIII.1.1 The reference time.....66
VIII.1.2 How to set a reference time 66
VIII.2 Time when outputting information 67
VIII.2.1 Plotting frames or vectors start time.....67
VIII.2.2 Plotting time on time series axis.....69

IX The Data Display as a data viewer 73
IX.1 Goals.....73
IX.2 The Data Display inside VEGA 73

IX.2.1 Displaying data from a frame channel source....73
IX.2.2 Interaction with the produced plots....73
IX.2.3 Using configuration files produced by the standalone dataDisplay application 74

X Signal processing and algorithms....75

XI The global variables often used....77
XI.1 The active manager gVM....77
XI.2 The active style gVStyle....77

XII Examples of macros....79
XII.1 Example 1 : Displaying a peak....79
XII.2 Example 2 : scrolling data....81
XII.3 Example 3 : histogramming some data and fitting the result....82
XII.4 Example 4 : Extracting some slow monitoring data and doing some simple plots....84
XII.5 Example 5 : Extracting a vector and plotting spectra....86

XIII How to compile your own code for VEGA or use the VEGA libraries in your code....89
XIII.1 Loading an existing shared library....89
XIII.2 Compiling a script and making a shared library of it....89
XIII.2.1 Preparation of the script....90
XIII.2.2 Compiling the script....90
XIII.2.3 Limitations....91
XIII.3 Building a standalone executable....91
XIII.3.1 Building an executable that is linked with the ROOT and VEGA libraries....91
XIII.3.2 Options of the vega-config utility....91
XIII.3.3 Using only a subset of capabilities....93
XIII.4 Building a general shared library usable with VEGA....93

I Appendix A : Local metadatabase structure and principles 95
I.1 Metadatabase for easy data access....95
I.1.1 Principle and structure....95
I.1.2 Creation of a metadatabase....96
I.1.3 Access to metadata....97
I.1.4 Getting general information about the metadatabase and it's contents....98
I.2 Dealing with selected or triggered data....99
I.2.1 Condition information in the metadatabase....99
I.2.2 Extracting metadata with a condition....100

Table of Contents....102